Reducing the Cost of Android Mutation Testing

Lin Deng
Department of Computer and Information Sciences
Towson University, Towson, Maryland
ldeng@towson.edu

Jeff Offutt
Department of Computer Science
George Mason University, Fairfax, Virginia
offutt@gmu.edu

***Abstract*—Due to the high market share of Android mobile devices, Android apps dominate the global market in terms of users, developers, and app releases. However, the quality of Android apps is a significant problem. Previously, we developed a mutation analysis-based approach to testing Android apps and showed it to be very effective. However, the computational cost of Android mutation testing is very high, possibly limiting its practical use. This paper presents a cost-reduction approach based on identifying redundancy among mutation operators used in Android mutation analysis. Excluding them can reduce cost without affecting the test quality. We consider a mutation operator to be *redundant* if tests designed to kill other types of mutants can also kill all or most of the mutants of this operator. We conducted an empirical study with selected open source Android apps. The results of our study show that three operators are redundant and can be excluded from Android mutation analysis. We also suggest updating one operator's implementation to stop generating trivial mutants. Additionally, we identity subsumption relationships among operators so that the operators subsumed by others can be skipped in Android mutation analysis.**

I. Introduction

Mobile applications (mobile apps) are software programs specifically developed for mobile devices. Due to the convenience of mobile devices, people use mobile apps more often than applications on other platforms [1]. Approximately 85% of mobile devices use the Android operating system [2]. In March 2018, more than 3.6 million Android apps are available for download on the Google Play Store [3]. However, many Android apps contain software faults, and users often experience problems. An Android analysis organization [3] found that 14% of Android apps are "low-quality."

Our prior work applied mutation testing to testing Android apps [4], explored the feasibility of Android mutation testing [5], and empirically evaluated its fault detection effectiveness using naturally occurring faults and crowdsourced faults [6]. The results show that Android mutation testing is very effective at detecting both types of software faults.

However, Android mutation testing can be expensive in several ways. Due to the constraints on size, weight, and power consumption, most Android devices are equipped with hardware that is slower than desktops and laptops. Executing and testing Android apps take more execution time than traditional software programs. Moreover, while Android mutation testing has been found to be effective at designing high-quality test cases and assessing test cases generated by other testing techniques, the number of mutants that need to be executed increases the cost of Android mutation testing. For example, executing 20 tests, one minute for each test, on 1,000 Android mutants may require up to 13.8 days.

DOI reference number: 10.18293/SEKE2018-184

This paper presents our experimental evaluation that tries to speed up mutation testing by finding redundant mutation operators that can be excluded from Android mutation testing, while still maintaining fault-detection effectiveness. Specifically, this experimental study analyzed redundancy among the 19 Java traditional mutation operators [7] and the 17 Android mutation operators [5].

The results of our study show that three mutation operators are redundant and can be excluded without reducing the effectiveness of Android mutation testing: (1) Unary Arithmetic Operator Deletion (AODU), (2) Unary Arithmetic Operator Insertion (AOIU), and (3) Logical Operator Insertion (LOI). Also, the design of the Activity Lifecycle Method Deletion (MDL) operator requires further improvement. Furthermore, the Button Widget Switch (BWS) operator subsumes Button Widget Deletion (BWD), and Operator Deletion (ODL) subsumes Constant Deletion (CDL), Conditional Operator Deletion (COD), and Variable Deletion (VDL). In addition, mutants created by the Fail on Back (FOB) operator, the TextView Deletion (TVD) operator, and the Orientation Lock (ORL) operator are very hard to kill.

This paper is organized as follows. Section II introduces background on Android mutation testing. Section III describes the experiment used to identify the redundancy among Android mutation operators, then, presents and analyzes the experiment results. Section IV gives an overview of related research. Section V discusses threats to validity, and the paper concludes and suggests future work in Section VI.

II. Background

In 1978, DeMillo et al. invented mutation testing [8], a syntax-based software testing technique that is very effective at designing high-quality tests and evaluating pre-existing tests. Mutation testing modifies a software artifact such as source code, to create new versions, called *mutants*. *Mutation operators* define the rules that specify the changes that are made to a software artifact. Testers design test cases to cause mutants to behave differently from the original, then the mutants are called *killed*. Well designed mutation operators can lead to very powerful test cases. The more mutants a test set can kill, the more effective the test set is at finding faults.

Mutation operators have been created for many different languages, including C and Java [7], [9]. Our prior work [4]–[6] used the novel programming features, unique characteristics, and testing challenges of Android apps to design and

TABLE I: Android Mutation Operators

Category	Android Mutation Operator
Event-based	Intent Payload Replacement (IPR)
	Intent Target Replacement (ITR)
	OnClick Event Replacement (ECR)
	OnTouch Event Replacement (ETR)
Component Lifecycle	Activity Lifecycle Method Deletion (MDL)
	Service Lifecycle Method Deletion (SMDL)
XML-related	Button Widget Deletion (BWD)
	EditText Widget Deletion (TWD)
	Activity Permission Deletion (APD)
	Button Widget Switch (BWS)
	TextView Deletion (TVD)
Common Faults	Fail on Null (FON)
	Orientation Lock (ORL)
	Fail on Back (FOB)
Context-aware	Location Modification (LCM)
Energy-related	WakeLock Release Deletion (WRD)
Network-related	Wi-Fi Connection Disabling (WCD)

evaluate 17 Android mutation operators, as listed in Table I. We also used 15 Java traditional method-level mutation operators [7] and four deletion mutation operators [10], [11].

III. Empirical Evaluation

Normally, all mutation operators are applied to generate mutants. This creates lots of mutants that must be executed many times at significant cost. Recent research [12] has found that between 90% and 99% of mutants are *redundant* in the sense that any test that kills another mutant is guaranteed to kill the redundant mutant. If redundant mutation operators can be identified and excluded, the cost of mutation will be significantly reduced. All mutants generated from the same mutation operator are of the same *type*. Thus, we use this study to determine whether mutants of one type are killed by the tests designed to kill mutants of other types.

In particular, this empirical evaluation tries to evaluate the redundancy in Android mutation testing by addressing the following research questions:

RQ1: How many mutants of one particular type can be killed by tests created to kill another type of mutants?

RQ2: Which types of mutants are less likely to be killed by tests created to kill other types of mutants?

RQ3: Can any mutation operator be excluded or improved without significantly reducing effectiveness?

A. Experimental Subjects

This experimental evaluation used 12 Android classes and their XML layout and configuration files from four open source Android apps: *JustSit* [13], *MunchLife* [14], *TippyTipper* [15], and *Tipster* [16]. Table II provides an overview of the projects. The 19 Java traditional mutation operators [7] generated 1,947 muJava mutants and the 17 Android operators generated 1,018 mutants. The number of muJava mutants ranged from four for About.java in TippyTipper to 534 for MunchLifeActivity.java in MunchLife, and the number of Android mutants ranged from one for AndroidManifest.xml in MunchLife to 258 for JustSit.java in JustSit.

B. Redundancy Scores

The mutation-adequate test set T_i includes tests that are specifically designed to kill all the mutation of type *i*. To quantify the redundancy among Java traditional mutation operators

TABLE II: Details of Experimental Subjects

Apps	Components	LOC	XML Nodes	muJava Mutants	Android Mutants
JustSit	JustSit.java	444		394	258
	main.xml		13		
	About.java	48		9	13
	about.xml		6		
	RunTimer.java	99		131	25
	run_timer.xml		3		
	JsSettings.java	61		28	31
	settings.xml		6		
	AndroidManifest.xml		14	0	4
Munch-Life	MunchLifeActivity.java	384		534	158
	main.xml		12		
	Settings.java	68		47	8
	preferences.xml		5		
	AndroidManifest.xml		10	0	1
Tippy-Tipper	TippyTipper.java	239		105	198
	main.xml		20		
	SplitBill.java	134		124	49
	SplitBill.xml		31		
	Total.java	279		231	115
	Total.xml		44		
	About.java	30		4	14
	About.xml		10		
	Settings.java	61		13	15
Tipster	TipsterActivity.java	297		327	129
	main.xml		30		
Total		**2144**	**204**	**1947**	**1018**

and Android mutation operators, Praphamontripong and Offutt [17] defined the redundancy score $r_{i,j}$ to be:

$$\text{Redundancy Score: } r_{i,j} = \frac{m_{i,j}}{M_j} \times 100\% \qquad (1)$$

where, $m_{i,j}$ is the number of mutants of type *j* killed by the mutation-adequate test set T_i, and M_j is the total number of non-equivalent mutants of type *j*.

In other words, the redundancy score $r_{i,j}$ is the percentage of mutants of type *j* killed by a test set that is adequate for type *i*. For example, a program has 100 non-equivalent Relational Operator Replacement (ROR) mutants and 200 non-equivalent Arithmetic Operator Insertion (AOIS) mutants. A tester designs a test set that kills all the non-equivalent AOIS mutants, getting an AOIS mutation-adequate test set. If this AOIS mutation-adequate test set also kills 60 ROR mutants, the redundancy score $r_{AOIS,ROR}$ in this program is $60 \div 100 = 60\%$.

Note that for a given subject app, according to the definition above, every possible pair of mutation operators has a redundancy score. Then, across all the subject apps in an experimental evaluation, there are multiple redundancy scores for the same pair of mutation operators with different values. For example, $r_{AOIS,ROR}$ may be 60% in subject s_1, 50% in s_2, and 40% in s_3. Consequently, a score that can represent the overall redundancy relationship is required. Praphamontripong and Offutt [17] defined the average redundancy score ($r_{average,i,j}$) to be the average value of all the $r_{i,j}$ of the operator in all experimental subjects. The average redundancy score is not weighted, i.e., we compute the redundancy score for each subject app, then calculate the average of the scores.

Redundancy score indicates quantitatively whether a mutation operator is redundant or not. For example, if a mutation operator has a redundancy score of 0%, it means no tests that were designed to kill other types of mutants killed any mutants of this type. That is, the mutation operator is not redundant.

However, if a mutation operator has a redundancy score of 100% for the tests that are specifically designed to kill mutants of another type, it means this operator is totally redundant and does not contribute anything to the quality of tests. Excluding it from the mutation analysis can reduce cost without reducing effectiveness. If a mutation operator has a redundancy score of 50%, half of the mutants generated by this operator are killed by the tests designed for other types of mutants. Some programs do not use all language features, thus the relevant mutation operators cannot be used to generate tests. Then, no tests will be designed for this mutation type.

C. Experimental Procedure

This study includes four steps to obtain the redundancy scores among the mutation operators:

1) **Generate mutants:** Given a subject, apply the 19 Java traditional mutation operators and the 17 Android mutation operators to to generate mutants. m_n represents the mutants created by operator *n*.
2) **Eliminate equivalent mutants and design tests:** For each set of mutants m_n, eliminate all equivalent mutants. Then, design a set of test cases to kill all the non-equivalent mutants, denoted by t_n, that is, tests designed to kill the mutants of type *n*. We design tests independently for each type of mutants. No redundant tests are introduced once all the mutants are killed.
3) **Execute tests:** For each set of test cases t_n, execute all tests on all mutants.
4) **Compute the redundancy scores:** For each pair of mutation operators and for each subject app, compute the redundancy score $r_{i,j}$. Then, to get an overview across all the subjects in the experiment, compute an average redundancy score for each mutant type.

Our tool implements a multithreading controller to parallelize the execution with multiple emulators and real devices. The tool executes on a MacBook Pro with a 2.6 GHz Intel i7 processor and 16 GB memory to control 8 emulators and 12 Motorola MOTO G Android smartphones in the experiment. All devices run on the Android KitKat operating system.

D. Experimental Results

This section presents experimental results and key findings.

RQ1: How many mutants of one particular type can be killed by tests created to kill another type of mutants?

Table III shows the average redundancy scores across all the subject apps. The columns represent mutation operators, and rows represent tests designed to kill all mutants of that type. So, for example, the tests designed to kill all AODU mutants ($test\_AODU$) killed 18.2% of the AOIS mutants. Some pairs mutant types never showed up in the same program, so their tests could not kill mutants of the other type. For example, LOR and CDL mutants never appeared together, so $test\_LOR$ is marked "n/a" for CDL, and vice versa.

Four Java traditional mutation operators, ASRS, LOD, SOR, and AODS, did not generate any mutants, and four Android mutation operators, ETR, LCM, SMDL, and WCD, did not generate any mutants. Thus, they are not listed in Table III. WRD mutants are also excluded because testers need to use the *dumpsys* tool to check system information, thus they cannot be redundant with other types of mutants. APD mutants are excluded because the principle of APD is to try all possible tests to identify those *un-killed* APD mutants, instead of designing tests to kill mutants.

RQ2: Which types of mutants are less likely to be killed by tests created to kill other types of mutants?

According to the results in Table III, three Android mutation operators were found to be very hard to kill. On average, only 6.4% of Fail on Back (FOB) mutants were killed by the mutation adequate test sets of other mutation operators, with the highest redundancy score of 33.3%. FOB injects a "Fail on Back" event handler into every Activity class. Since Android apps are event-based programs, their execution flows rely heavily on events initiated by user actions. The Back button lets users move backward to the previous Activity, interrupting the usual execution flow. It is usually not on the "happy path" from the perspective of software design, and results in a common fault of Android apps, that is, the software fails when the Back button is clicked. To kill FOB mutants, testers need to design tests that press the *Back* button at least once at every Activity. However, in this experiment, very few tests designed for other mutation operators included the user action of clicking the *Back* button.

Very few TextView Deletion (TVD) mutants were killed. On average, less than 1% of TVD mutants were killed by the mutation adequate test sets of other mutation operators, and its highest redundancy score was 8.3%. Since TextView widgets cannot be edited by users, they usually do not associate with any user events, nor require event handlers from the implementation of the app. However, TextView widgets are widely used by developers to present essential information. TVD deletes TextView widgets from screens one at a time. Killing a TVD mutant needs a test to ensure that the TextView widget displays correct information. Very few tests checked TextView widgets' contents, unless the TextView widget was used to display some variable results, such as a tip amount.

Very few Orientation Lock (ORL) mutants were killed. On average, only 2.5% of Orientation Lock (ORL) mutants were killed by the mutation adequate test sets of other mutation operators, and its highest redundancy score was 12.5%. Most mobile devices have the unique feature of being able to change the screen orientation. To use to this feature, many apps change their layout of the GUI when the orientation changes. However, different screen sizes and resolutions on different devices make switching the orientation difficult for the developers, leading to faults. ORL mutants freeze the orientation of an Activity by inserting a special *locking* statement into the source code, so that no switching actions can be accepted by the app. To kill ORL mutants, testers need to design tests that explicitly change the orientation, then check whether the GUI structure is displayed as expected after switching the orientation. In this experiment, no other mutation operators consider switching the screen orientation, so there was no redundancy.

RQ3: Are any Android mutation operators redundant enough to be excluded, or can any be improved? In particular, can the mutants of one type always be killed by tests created to kill another type?

TABLE III: Average Redundancy Scores

	muJava Mutation Operator														Android Mutation Operator										
	AODU	**AOIS**	**AOIU**	**AORB**	**CDL**	**COD**	**COI**	**COR**	**LOI**	**LOR**	**ODL**	**ROR**	**SDL**	**VDL**	**BWD**	**BWS**	**ECR**	**FOB**	**FON**	**IPR**	**ITR**	**MDL**	**ORL**	**TVD**	**TWD**
test_AODU	——	0.182	0.351	0.250	0.000	0.000	0.108	0.000	0.286	0.000	0.053	0.169	0.165	0.000	0.125	0.333	0.200	0.000	0.917	0.000	0.167	0.750	0.000	0.000	0.500
test_AOIS	1.000	——	0.819	0.875	0.750	1.000	0.537	0.524	0.631	0.000	0.491	0.504	0.545	0.937	0.688	0.111	0.869	0.167	0.567	0.800	0.778	0.875	0.021	0.000	0.667
test_AOIU	1.000	0.545	——	0.865	0.600	0.750	0.466	0.476	0.681	0.000	0.402	0.443	0.560	0.916	0.813	0.167	0.869	0.125	0.472	0.800	0.944	0.781	0.016	0.000	0.667
test_AORB	1.000	0.563	0.688	——	0.750	0.750	0.500	0.643	0.561	0.000	0.527	0.429	0.396	0.947	0.688	0.111	0.869	0.000	0.458	0.800	0.778	0.850	0.025	0.000	0.667
test_CDL	1.000	0.601	0.768	0.633	——	0.750	0.456	0.643	0.645	n/a	0.567	0.497	0.546	0.702	0.583	0.111	0.803	0.000	0.333	n/a	1.000	0.900	0.025	0.050	0.500
test_COD	1.000	0.211	0.632	0.333	1.000	——	0.500	0.286	0.500	n/a	0.474	0.326	0.342	0.333	0.250	0.333	n/a	0.000	1.000	n/a	n/a	1.000	0.000	0.000	0.000
test_COI	1.000	0.424	0.471	0.613	0.500	1.000	——	0.905	0.574	0.500	0.439	0.854	0.621	0.579	0.833	0.417	0.775	0.200	0.708	0.800	0.958	0.850	0.025	0.000	0.500
test_COR	0.000	0.412	0.385	0.075	0.125	0.750	0.874	——	0.505	n/a	0.277	0.725	0.563	0.026	0.375	0.333	0.333	0.333	0.667	n/a	1.000	0.833	0.042	0.000	0.000
test_LOI	1.000	0.777	0.868	0.765	0.600	1.000	0.461	0.571	——	0.000	0.425	0.645	0.590	0.916	0.813	0.278	0.869	0.125	0.472	0.800	0.778	0.781	0.016	0.000	0.667
test_LOR	1.000	0.286	0.176	0.500	n/a	n/a	0.105	n/a	0.138	——	0.000	0.152	0.132	0.000	1.000	n/a	0.600	0.000	0.833	0.800	0.333	0.500	0.000	0.000	1.000
test_ODL	1.000	0.777	0.847	0.885	1.000	1.000	0.689	1.000	0.796	0.500	——	0.712	0.673	1.000	0.792	0.278	0.869	0.143	0.567	0.800	0.972	0.893	0.018	0.042	0.667
test_ROR	1.000	0.654	0.543	0.838	1.000	1.000	0.900	1.000	0.621	0.500	0.664	——	0.694	0.658	1.000	0.417	1.000	0.200	0.708	0.800	0.958	0.850	0.125	0.000	0.500
test_SDL	1.000	0.730	0.921	0.920	1.000	1.000	0.926	1.000	0.853	0.500	0.916	0.945	——	0.937	1.000	0.278	1.000	0.250	0.639	0.800	0.972	0.906	0.078	0.083	1.000
test_VDL	1.000	0.706	0.743	0.708	1.000	0.750	0.368	0.643	0.653	0.500	0.527	0.429	0.424	——	0.688	0.111	0.869	0.000	0.458	0.400	0.972	0.850	0.025	0.000	0.667
test_BWD	1.000	0.401	0.566	0.431	0.333	0.750	0.561	0.786	0.583	0.000	0.316	0.459	0.331	0.355	——	0.278	1.000	0.000	0.458	0.800	0.833	0.688	0.031	0.000	0.000
test_BWS	0.000	0.009	0.445	0.033	0.000	0.000	0.200	0.000	0.418	n/a	0.013	0.048	0.185	0.018	1.000	——	0.000	0.000	0.333	n/a	0.000	0.917	0.000	0.000	0.000
test_ECR	1.000	0.500	0.532	0.575	0.750	n/a	0.453	1.000	0.521	0.000	0.400	0.397	0.470	0.474	1.000	0.000	——	0.000	0.278	1.000	0.972	0.583	0.042	0.000	0.500
test_FOB	0.500	0.004	0.503	0.020	0.000	0.000	0.080	0.000	0.449	0.000	0.006	0.049	0.312	0.011	0.000	0.000	0.000	——	0.472	0.000	0.000	0.281	0.000	0.000	0.000
test_FON	0.500	0.138	0.578	0.363	0.125	0.000	0.276	0.333	0.537	0.000	0.125	0.246	0.338	0.079	0.500	0.000	0.652	0.000	——	0.400	0.639	0.607	0.018	0.000	0.000
test_IPR	1.000	0.286	0.353	0.500	n/a	n/a	0.105	n/a	0.138	0.000	0.389	0.152	0.388	0.000	1.000	n/a	1.000	0.000	0.833	——	0.500	0.500	0.000	0.000	0.000
test_ITR	1.000	0.262	0.471	0.367	0.250	n/a	0.186	0.000	0.415	0.000	0.240	0.135	0.251	0.035	0.389	0.000	0.563	0.000	0.278	0.800	——	0.750	0.000	0.000	0.500
test_MDL	0.500	0.062	0.520	0.040	0.250	0.000	0.171	0.333	0.472	0.000	0.023	0.109	0.334	0.021	0.125	0.000	0.111	0.000	0.472	0.000	0.333	——	0.016	0.000	0.000
test_ORL	0.000	0.007	0.503	0.020	0.000	0.000	0.080	0.000	0.449	0.000	0.006	0.019	0.291	0.011	0.000	0.000	0.000	0.000	0.472	0.000	0.000	0.219	——	0.000	0.000
test_TVD	0.500	0.010	0.449	0.025	0.200	0.000	0.100	0.000	0.426	0.000	0.223	0.062	0.363	0.211	0.000	0.000	0.000	0.000	0.458	0.000	0.000	0.833	0.083	——	0.000
test_TWD	0.500	0.095	0.518	0.167	0.000	0.250	0.192	0.143	0.476	0.000	0.147	0.157	0.183	0.000	0.167	0.333	0.603	0.000	0.611	0.000	0.500	0.833	0.000	0.000	——
Average	**0.771**	**0.360**	**0.569**	**0.450**	**0.465**	**0.538**	**0.387**	**0.468**	**0.514**	**0.125**	**0.319**	**0.361**	**0.404**	**0.382**	**0.576**	**0.177**	**0.602**	**0.064**	**0.561**	**0.530**	**0.626**	**0.743**	**0.025**	**0.007**	**0.375**

Excluding:	
AODS	APD
ASRS	ETR
LOD	LCM
SOR	SMDL
	WCD
	WRD

According to the results, several mutation operators generated mutants that were easily killed by the tests designed to kill other types of mutants. Among the 17 Android mutation operators, the Activity Lifecycle Method Deletion (MDL) mutation operator has the highest mean redundancy score (74.3%). Android operating systems require all components in Android apps to behave according to a pre-defined lifecycle. If developers want to define a specific behavior when an Activity switches its state, they must follow the lifecycle and override correct methods in it. For example, after an Activity is launched, three methods, *onCreate()*, *onStart()*, and *onResume()*, need to be executed sequentially before the user can see the Activity on the screen. MDL deletes each overriding method to force Android to call the version in the super class. This requires the tester to design tests that ensure the app is in the correct expected state. However, many developers use *onCreate()* to define and initialize GUI structures of their apps. After MDL deletes the content of *onCreate()*, no GUI widgets can be displayed for the current Activity. Then any test case that looks for a GUI widget or initiates a user event can kill MDL mutants.

A recommendation is that instead of simply deleting the content of *onCreate()*, an alternative implementation is to move the content of *onCreate()* to *onStart()* and *onResume()*. Figure 1 gives an example of the recommended implementation. All the code that defines GUI widgets and initializes event handlers has been migrated from *onCreate()* to *onStart()*. In this way, MDL mutants are no longer trivial. In addition, the only way to kill this new version of MDL mutants is to make the Activity switch among different states, so that different lifecycle methods can be called. Therefore, modified MDL would require testers to design tests to make the Activity switch among different states.

The Unary Arithmetic Operator Deletion (AODU) mutation operator has the highest mean redundancy score (77.1%) of the 19 muJava mutation operators [7]. 16 sets of mutation adequate test sets designed to kill other types of mutants killed

```
public void onCreate (Bundle savedInstanceState) {
   super.onCreate (savedInstanceState);
   setContentView (R.layout.main);
   Button up_button = (Button) findViewById (R.id.up_button);
}
public void onStart () {
   super.onStart ();
}
```

Original

```
public void onCreate (Bundle savedInstanceState) {
   super.onCreate (savedInstanceState);
}
public void onStart () {
   super.onStart ();
   setContentView (R.layout.main);
   Button up_button = (Button) findViewById (R.id.up_button);
}
```

Mutant

Fig. 1: Recommended Implementation of MDL

all AODU mutants, indicated by "1.000" values in the AODU column in Table III. AODU deletes basic unary arithmetic operators in an expression. Figure 2 shows an example AODU mutant, in which the minus symbol is deleted. The results indicate that AODU is redundant and can be excluded.

Original: int x = - y ;	AODU Mutant: int x = y ;

Fig. 2: An Example AODU Mutant

As shown in Table III, the Button Widget Deletion (BWD) column has six "1.000" values, which is the second highest among all the mutation operators. In fact, all the BWD mutants were killed by the BWS tests. Button widgets are used by nearly all Android apps in many ways. BWD deletes buttons one at a time from the XML layout file of the UI. BWS switches the locations of two buttons on the same screen. In this way, the function of a button is unaffected, but the GUI layout looks different from the original version. BWS requires the tester to design tests that deliberately check the location (either relative or absolute) of a button widget.

When BWS mutants ensure every button is displayed at the expected location, it also guarantees that this button is shown

on the screen. *Subsumption* is used to theoretically compare test criteria: a criterion C1 *subsumes* another criterion C2, if every test that satisfies C1 is guaranteed to satisfy C2 [18]. In mutation testing, an operator MO1 *subsumes* another operator MO2 if a test set that kills all mutants of MO1 is guaranteed to kill all mutants of MO2. Thus, BWS subsumes BWD, that is, every test set designed to kill all the BWS mutants can kill all the BWD mutants. As a result, when users include BWS in the Android mutation analysis, excluding BWD mutants will **not** affect test effectiveness. Note that if an Activity only has one button widget, BWS cannot generate any mutants. This is because to achieve *switching*, the Activity must display at least two buttons. Thus, it is recommended to disable BWD when there are BSW mutants, and enable it otherwise.

The Conditional Operator Deletion (COD) mutation operator also has six "1.000" values (second highest), and the Constant Deletion (CDL) mutation operator has five "1.000" values (third highest). Also, the ODL test sets killed all the mutants of CDL, COD, and the Variable Deletion mutation operator (VDL). The Operator Deletion mutation operator (ODL) was originally designed by Delamaro et al. [11]. It deletes each arithmetic, relational, logical, bitwise, and shift operator from all expressions. CDL deletes each constant in an expression, and VDL deletes each variable in an expression. Figures 3 shows example ODL, CDL, and VDL mutants. According to the definitions, it is guaranteed that ODL subsumes CDL and VDL. COD deletes unary conditional operators. Figure 3 also shows that ODL and COD generate the same mutants. Therefore, ODL theoretically subsumes COD. Not surprisingly, test cases designed to kill ODL mutants also kill CDL, COD, and VDL mutants, which means when using ODL, we can exclude CDL, COD, and VDL.

Original: int x = y + 2 ;	ODL Mutant_1: int x = y ; ODL Mutant_2: int x = 2 ; CDL Mutant: int x = y ; VDL Mutant: int x = 2 ;
Original: int x = - y ;	ODL Mutant: int x = y ;
Original: if (! isError) { x = y ; }	ODL Mutant: if (isError) { x = y ; } COD Mutant: if (isError) { x = y ; }

Fig. 3: Example ODL, CDL, VDL, and COD Mutants

The Unary Arithmetic Operator Insertion (AOIU) inserts a minus sign in front of integer variables. The Logical Operator Insertion (LOI) inserts a bitwise complement operator in front of integer variables. 50.3% of AOIU mutants and 44.9% of LOI mutants were killed by test_FOB tests, which are simple tests that only launch an Activity and click the Back button.

```
int level = 1;
current_level.setText ( Integer.toString ( level ) ); // Original
int level = 1;
current_level.setText ( Integer.toString ( -level ) ); // AOIU Mutant
int level = 1;
current_level.setText ( Integer.toString ( ∼level ) ); // LOI Mutant
```

Fig. 4: AOIU and LOI Examples

Figure 4 gives example AOIU and LOI mutants. In Android apps, each GUI widget is assigned a resource ID that is recorded as an integer number. These resource IDs are stored and managed in XML files. Both AOIU and LOI generate many mutants by mutating the resource IDs in Android apps. Figure 5 shows an example where AOIU changes the resource ID of *upbutton*. However, once a resource ID is changed and not mapped to its original GUI widget, the Android app will immediately crash after launched, making the mutant trivial and redundant. That is, any test case that launches the app can kill this mutant. Similarly, LOI also generates trivial mutants. Therefore, when using mutation testing for Android apps, we recommend to exclude AOIU and LOI.

```
Button upbutton = (Button) findViewById ( R.id.upbutton ); // Original
Button upbutton = (Button) findViewById ( - R.id.upbutton ); // AOIU
```

Fig. 5: AOIU Changes Android Resource ID

In summary, we recommend the following:

1) Exclude AODU, because of its highest average redundancy scores
2) Improve the design of MDL, because MDL generates trivial mutants
3) Exclude BWD when using BWS, because BWS subsumes BWD
4) Exclude AOIU and LOI, because around 50% of AOIU and LOI mutants are trivial
5) Exclude CDL, COD, and VDL when using ODL, because ODL subsumes them

E. Re-evaluating the Effectiveness

Based on the evaluation results, we provide recommendations to eliminate the redundancy among Android mutation operators. However, it is not clear whether the effectiveness of Android mutation testing still holds after removing and modifying redundant mutation operators. Due to the high computational cost of Android mutation testing, re-conducting the whole effectiveness evaluation in Section III would take several months. Thus, we elected to check the results on one subject app.

According to the recommendations, we updated the implementation of our Android mutation testing tool. We took Tipster as the subject app for the re-evaluation. Originally, Tipster generated 327 muJava mutants and 130 Android mutants. After removing and modifying redundant mutation operators, Tipster generated 259 muJava mutants and 125 Android mutants, with an overall 16% reduction in terms of the total number of the mutants. After that, a new set of mutation adequate tests was designed. Originally, Tipster had 64 crowdsourced faults, in which 51 were detected by the old mutation adequate test set. After re-conducting the evaluation, the newly designed mutation adequate test set using fewer and less redundant mutants found the same 51 crowdsourced faults in Tipster. Therefore, it is concluded that removing and modifying redundant mutation operators in this research did not impact the effectiveness of Android mutation testing.

IV. Related Work

Traditional mutation testing uses three types of approaches to reduce cost: *do-fewer*, *do-smarter*, and *do-faster* [19]. As a *do-fewer* approach, *selective mutation* was introduced by Wong and Mathur to choose a subset of mutation operators [20]. The muJava tool selects 15 operators to preserve almost the same test coverage as non-selective mutation [7]. Empirical studies in both Java and C show that the Deletion mutation operators are able to result in very effective tests with much lower cost [10], [11]. This study, as a *do-fewer* approach, also discussed them in Android mutation testing.

V. Threats to Validity

Similar to most experiments in software engineering, this empirical evaluation has several threats to validity.

Internal validity: In this experiment, we designed only one set of Android mutation-adequate tests for each type of mutant. The results of redundancy scores may differ for different Android mutation-adequate tests. Also, in this experimental study, we identified all the equivalent mutants by hand. Manual work could introduce human errors.

External validity: We cannot guarantee that the selected subjects are representative. The results and redundancy scores may differ from the results in this study if we used different subject apps. To improve the ability to compare results, we chose Android apps that have previously been used in other Android testing studies.

Construct validity: The implementation of our Android mutation testing tool and the associated mutation operators may include software faults. In this study, we constantly tested the experimental environment to ensure reliability.

VI. Conclusions and Future Work

Android mutation testing is an effective approach to design and evaluate tests for Android apps. However, due to the unique conditions of Android devices and apps, the cost of Android mutation testing can be very expensive, in terms of computational time and effort. We conducted an empirical study to identify redundancy among mutation operators, with the goal of finding mutation operators that are redundant and do not contribute to the quality of tests.

The results of our study show that three Java traditional mutation operators (AODU, AOIU, and LOI) are redundant in Android mutation analysis. Excluding them can save costs without reducing test quality. As BWS subsumes BWD, we recommend skipping BWD mutants when BWS is used. As ODL subsumes CDL, COD, and VDL, these three can be excluded if ODL is used. Our study indicates that three Android mutation operators (FOB, TVD, and ORL) have very low average redundancy scores (6.4%, 0.7%, and 2.5%). They are very hard to kill by other types of tests. Also, we provide a recommendation for improving the design of MDL to stop generating trivial mutants.

Kurtz et al. [21] found that traditional mutation scores are inflated during mutation analysis, so are flawed as a test quality measurement device. Since a very strong and rich test set is needed to perform minimal mutation analysis and compute dominator mutation scores, we did not include them into this study, due to the expensive cost. For future work, we hope to use minimal mutation analysis and dominator mutation scores to verify the conclusions in this study.

Acknowledgment

This work was partly funded by The Knowledge Foundation (KKS) through the project 20130085: Testing of Critical System Characteristics (TOCSYC).

References

[1] Kleiner Perkins Caufield & Byers, "Internet trends 2015," Online, May 2015, http://www.kpcb.com/internet-trends, last access September 2015.

[2] International Data Corporation, "Smartphone OS market share, 2017 Q1," Online, May 2017, https://www.idc.com/promo/smartphone-market-share/os, last access March 2018.

[3] "Android apps on Google Play," 2018, http://www.appbrain.com/stats/number-of-android-apps, last access March 2018.

[4] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of Android apps," in *Tenth Workshop on Mutation Analysis (Mutation 2015)*, April 2015, pp. 1–10.

[5] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Information and Software Technology*, vol. 81, pp. 154 – 168, 2017.

[6] L. Deng, J. Offutt, and D. Samudio, "Is mutation analysis effective at testing android apps?" in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2017, pp. 86–93.

[7] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava : An automated class mutation system," *Software Testing, Verification, and Reliability, Wiley*, vol. 15, no. 2, pp. 97–133, June 2005.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[9] H. Agrawal, R. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and G. Spafford, "Design of mutant operators for the C programming language," Software Engineering Research Center, Purdue University, West Lafayette, IN, Technical Report SERC-TR-41-P, March 1989.

[10] L. Deng, J. Offutt, and N. Li, "Empirical evaluation of the statement deletion mutation operator," in *6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, Luxembourg, March 2013.

[11] M. E. Delamaro, J. Offutt, and P. Ammann, "Designing deletion mutation operators," in *7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*, Cleveland, Ohio, March 2014.

[12] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*, Cleveland, OH, March 2014, pp. 21–30.

[13] (2010) JustSit. https://play.google.com/store/apps/details?id=com.brocktice.JustSit, last access September 2016.

[14] (2014) MunchLife. https://play.google.com/store/apps/details?id=info.bpace.munchlife, last access September 2016.

[15] (2013) TippyTipper. https://code.google.com/p/tippytipper, last access September 2016.

[16] I. Darwin, "Tipster," 2016, https://github.com/IanDarwin/Android-Cookbook-Examples/tree/master/Tipster, last access September 2016.

[17] U. Praphamontripong and J. Offutt, "Finding redundancy in web mutation operators," in *Twelfth Workshop on Mutation Analysis (Mutation 2017)*, March 2017, pp. 134–142.

[18] P. Ammann and J. Offutt, *Introduction to software testing*, 2nd ed. Cambridge University Press, 2017, iSBN 978-1107172012.

[19] J. Offutt and R. Untch, "Mutation 2000: Uniting the orthogonal," in *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, CA, October 2000, pp. 45–55.

[20] W. E. Wong, M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Constrained mutation in C programs," in *Proceedings of the 8th Brazilian Symposium on Software Engineering*, Curitiba, Brazil, October 1994, pp. 439–452.

[21] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the validity of selective mutation with dominator mutants," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 571–582.