FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Assembly-Level Mutant Schemata for Android Applications

João das Neves Fernandes

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. João Bispo

Second Supervisor: Prof. Auri Vincenzi

July 10, 2026

Assembly-Level Mutant Schemata for Android Applications

João das Neves Fernandes

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Bruno Lima
Examiner: Prof. João Saraiva
Member: Prof. João Bispo

July 10, 2026

Resumo

O teste de mutação é uma técnica para avaliar a qualidade de conjuntos de testes de software, introduzindo pequenas falhas deliberadas num programa e verificando se os testes as detetam. Aplicar o teste de mutação a aplicações Android é desafiante porque as ferramentas existentes exigem normalmente acesso ao código-fonte Java ou Kotlin e são difíceis de utilizar em projetos reais que misturam linguagens ou incluem componentes de código fechado.

Este trabalho apresenta o metford-apk, uma ferramenta de teste de mutações para Android que opera diretamente no bytecode Dalvik. A ferramenta implementa a estratégia de schemata, incorporando todos os mutantes num único APK através de um switch em tempo de execução e fornece catorze operadores de mutação que abrangem falhas aritméticas e padrões específicos do Android relacionados com a comunicação de Intent e o comportamento da interface gráfica.

O metford-apk é avaliado em comparação com o METFORD, uma ferramenta de teste de mutação de schemata ao nível do código-fonte, em seis aplicações Android de código aberto. Os resultados indicam que o metford-apk gera um número comparável ou superior de mutantes em todos os sujeitos e cobre entre 57% e 81% dos locais de mutação identificados pelo METFORD aos níveis de classe e operador. Os tempos de compilação são competitivos com a abordagem ao nível do código-fonte. A ferramenta é também aplicada a três aplicações de código fechado, demonstrando a sua aplicabilidade para além do alcance das ferramentas de nível de código-fonte.

Palavras-chave: Teste de Mutação • Android • Smali • Schemata • Engenharia de Software

Abstract

Mutation testing is a technique for evaluating the quality of software test suites by introducing small, deliberate faults into a program and checking whether the tests detect them. Applying mutation testing to Android applications is challenging because existing tools typically require access to Java or Kotlin source code and are difficult to use on real projects that mix languages or include closed-source components.

This work presents metford-apk, a mutation testing tool for Android that operates directly on Dalvik bytecode. The tool implements the schemata strategy, embedding all mutants in a single APK through a runtime-selected switch dispatch, and provides fourteen mutation operators covering arithmetic faults and Android-specific patterns related to Intent communication and user interface behaviour.

metford-apk is evaluated against METFORD, a source-level schemata mutation testing tool, across six open-source Android applications. The results show that metford-apk generates a comparable or greater number of mutants in all subjects and covers between 57% and 81% of the mutation sites identified by METFORD at the class and operator level. Build times are competitive with the source-level approach. The tool is also applied to three closed-source applications, demonstrating its applicability beyond the reach of source-level tools.

Keywords: Mutation Testing • Android • Smali • Schemata • Software Engineering

The United Nations Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals (https://sdgs.un.org/goals) to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

The specific Sustainable Development Goals for this thesis have the following names:

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialisation and foster innovation

SDG	Target	Contribution	Performance Indicators and Metrics
9	9.5	metford-apk advances software quality assurance research by enabling mutation testing on Android applications without requiring source code access, broadening the reach of automated testing tools to a wider range of software systems.	Increase in the range of Android applications amenable to mutation testing; number of mutation operators applicable at the assembly level.
	9.b	Supporting the development of open-source software testing tools contributes to domestic technology development and industrial diversification in the software sector.	Availability of open-source tooling for Android mutation testing.

Acknowledgements

I would like to thank my supervisors for the opportunity, guidance, and patience. They answered all the questions I had and guided me through all the work I needed to do. I also thank their availability to help me whenever I needed, given my constraints as I was working in parallel to the dissertation.

I would also like to thank my university friends and colleagues, who made these last few years some of the best of my life, and who reinforced my passion for computers with their own.

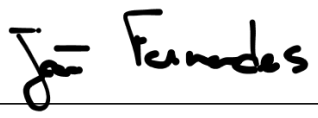
Lastly, I would like to thank my family for their support and patience throughout my life, and for giving me all the opportunities I have had.

João das Neves Fernandes

Declaration of honour

I declare that this dissertation is my own work and has not previously been submitted or assessed at this or any other institution. References to other authors (statements, ideas, thoughts), as well as the use of published works, strictly comply with the rules of attribution and are duly identified in the text and in the bibliographic references, in accordance with the applicable referencing standards. I am aware that the practice of plagiarism or self-plagiarism constitutes an academic offence, as established in the U.Porto Code of Ethics.
I also declare that I have used Claude to help with coding and writing of this dissertation, and that this use and its results are duly noted and have been carefully checked for errors, inaccuracies, unfounded attributions or other forms of bias in content and arguments, as well as determining authorship.

João das Neves Fernandes



"There are two days in the year that we can not do anything, yesterday and tomorrow"

Mahatma Gandhi

Contents

List of Tables

List of Acronyms

AST	Abstract Syntax Tree
PFP	Potential Faults Profile

Chapter 1

Introduction

1.1 Context

Modern life has become increasingly reliant on mobile applications, which facilitate a variety of tasks like shopping, entertainment, and communication. According to Statista [27], mobile app downloads continue to rise, reaching approximately 257 billion in 2023. This increase demonstrates the need for better ways to ensure software quality.

One of the most common ways to verify software quality is through test sets. These can ensure that the applications behave as expected and that no unpredictable behavior occurs on the user's end.

Mutation testing is a method used to evaluate how effective software tests are. It works by making small changes to a program to imitate common programming mistakes. These modified versions are called *mutants*. If the test suite detects the change, the mutant is said to be *killed*; if not, it *survives*, revealing a weakness in the tests. For example, if a mutation swapped the operator `+` with `-` in an `add()` function and the tests still passed, this would mean the tests were not verifying the function's output correctly. This process helps measure and improve the quality of test sets [15].

However, one of the drawbacks of mutation testing can be the computational cost of running multiple mutations, which, if not careful, can be very expensive. This is because, in traditional mutation testing, we create a new program for each mutation that we want to run.

A solution for this problem can be the *schemata* strategy for mutation testing, which combines all mutation operations into a single version of the program by using conditional statements, such as `if` or `switch`. This approach, also known as the *metamutant* strategy, reduces the need to create and execute multiple program versions, which helps to lower storage use and execution time [32].

1.2 Motivation

Although mutation testing has been studied for many years, it remains underutilized in Android development. The main problem is not that the results are weak, but rather that the tools are too complex to use effectively. Android projects often utilize both Java and Kotlin, rely on complex build systems, and incorporate numerous external libraries.

Most existing mutation testing tools work at the source code level. They require full access to the source and are typically designed for a single language. Moreover, we also need to pass through the compilation process to create the APK to be able to run the application with the mutations. All this motives make them difficult to use in real projects that mix languages or are partly closed-source. In many cases, such as when testing third-party libraries or studying closed-source apps, only the compiled APK is available. This limits the use of source-level tools in mobile application testing.

A solution to this problem is applying mutation operations directly into an assembly code, smali, which can be generated directly from the bytecode. There are already some implementations of this solution, but with some limitations, such as high computational cost and a small number of supported mutation operators [13].

1.3 Problem Statement

Existing approaches that apply mutation testing directly to Android APKs have shown that it is possible to perform this process without access to the source code. However, these approaches still face limitations. At the same time, most traditional mutation testing strategies were developed for source code and are not easily transferred to assembly-level representations.

This creates an open question about the extent to which source-level mutation concepts, including specific operators and the *schemata* strategy, can be applied at the assembly-level. Since not all operators in the source code have direct equivalents in assembly, some adaptation or transformation may be required. Their effectiveness can be analyzed by comparing results obtained from APKs produced at the smali level and those produced at the Java level.

The *schemata* approach also has some drawbacks at the source-level, such as limits on where conditional logic can be used and difficulties in applying some mutation operators. One example of this can be that control flow operators like if and switch can only be used inside methods. These drawbacks can also be studied at the assembly-level, where the code structure and control flow are different.

1.4 Research Questions

The following research questions guide this work:

RQ1 How can source-level mutation-testing strategies be applied at the assembly-level?

RQ2 Which source-level mutation operators remain meaningful when applied at assembly-level?

RQ3 Does assembly-level improve the efficiency and applicability of mutation testing compared to source-level?

1.5 Hypothesis

Considering the problem previously identified, this work formulates the following hypothesis:

It is possible to improve mutation testing efficiency and applicability on Android by implementing its strategies at assembly-level.

1.6 Contributions

The main contribution of this work is a schemata mutation testing approach that operates directly at the Android assembly level, without requiring access to source code. This includes a method for encoding the mutant schemata strategy in Dalvik bytecode using `packed-switch` dispatch and an adaptation of fourteen Android-specific mutation operators from the METFORD operator set to the smali representation. The approach is realised as a prototype tool, metford-apk, and evaluated against METFORD on six open-source Android applications as well as three closed-source applications that source-level tools cannot process.

1.7 Document Structure

The remainder of this document is organised as follows. Chapter 2 introduces the background concepts necessary to understand this work, including mutation testing fundamentals and the tool used to manipulate Android assembly. Chapter 3 surveys related work on mutation testing tools and approaches, with a focus on Android and bytecode-level techniques. Chapter 4 describes the design and implementation of metford-apk, covering the schemata encoding, the mutation controller, the operator adaptations, and the configuration and reporting mechanisms. Chapter 5 presents the evaluation of the tool against METFORD across a set of Android applications. Chapter 6 concludes the work and outlines directions for future investigation.

Chapter 2

Background

2.1 Mutation Testing

Mutation testing is a fault-based testing technique used to assess the effectiveness of a test suite. The main idea is to create modified versions of a program, called mutants, by introducing small syntactic changes that simulate typical programming faults. A test suite is then executed against these mutants, and its quality is measured by its ability to detect the injected faults. If a test causes different behavior on a mutant than on the original program, the mutant is considered killed.

Originally proposed for procedural and object-oriented programs, mutation testing has been shown to be a strong test adequacy criterion, often outperforming traditional coverage metrics in terms of fault detection capability [15]. Over the years, mutation testing has been applied to different languages and paradigms, and several tools have been developed to automate mutant generation and execution.

Despite its effectiveness, mutation testing is known for its high computational cost, mainly due to the large number of mutants that can be generated and the need to execute tests for each of them. As a result, much of the research in this area has focused on reducing cost while preserving fault detection capability. These challenges become more pronounced in complex platforms such as Android, where applications rely on event-driven execution, multiple components, and resource-constrained environments.

2.2 LARA Framework

LARA [1] is a source-to-source code analysis and transformation framework developed at the SPeCS research lab, originally designed as an aspect-oriented domain-specific language and later evolved into a general-purpose framework [23]. It allows developers and researchers to analyze, instrument, and transform programs by writing scripts, currently in JavaScript or TypeScript, that operate over structured representations of source code. LARA focuses particularly on non-functional

[1] https://github.com/specs-feup/lara-framework

concerns such as performance, energy efficiency, and code instrumentation, and it underpins several language-specific weavers (e.g., Clava for C/C++, Kadabra for Java) that allow the same transformation logic to be reused across different programming languages while keeping the source code itself as the main interface [1, 2].

2.2.1 Alpakka

Alpakka is a source-to-source compiler for Android applications that applies the LARA framework to smali, the assembly-like language representing Dalvik bytecode inside APK files. Its main goal is to enable systematic analysis and transformation of Android applications without requiring access to the original source code or depending on the language or framework used during development. By operating directly on smali, Alpakka preserves full fidelity with the compiled application and ensures that all transformations remain recompilable into valid APKs [25].

In practice, Alpakka takes an APK as input, decompiles it into smali, and constructs a structured intermediate representation that models application elements such as classes, methods, instructions, and references. This representation abstracts away much of the complexity of raw smali while still exposing low-level details necessary for analysis. On top of this model, Alpakka provides control-flow information that captures jumps, conditionals, exceptions, and Android-specific callback behavior, allowing analyses to reason about how execution proceeds through different paths and component lifecycles.

Chapter 3

State of the Art

3.1 Mutation Operators

Mutation operators define the types of faults that are injected during mutation testing. In the context of Android applications, traditional operators designed for general-purpose languages are not sufficient to capture the specific characteristics of the platform. Android apps rely heavily on component interactions, lifecycle callbacks, XML-based configuration, and context-dependent behavior. As a result, several works propose Android-specific mutation operators that target these aspects explicitly.

Early work by Deng et al. introduced one of the first systematic sets of mutation operators for Android applications [9]. Their operators extend traditional Java mutation with Android-specific faults, focusing on event handling, component lifecycles, GUI elements, and configuration files. This work laid the foundation for many later tools and operator catalogs.

A common class of Android mutation operators targets intents and inter-component communication. Intents are central to Android applications and are used to trigger activities, services, and broadcast receivers. Operators in this category typically modify intent payloads, replace target components, or alter intent filters. For example, intent payload replacement mutates the data passed between components, while intent target replacement redirects an intent to a different compatible component [9]. These operators model faults where incorrect data is propagated or where the wrong component is invoked.

Another important category focuses on component lifecycle management. Android components follow predefined lifecycles, and incorrect handling of lifecycle callbacks is a frequent source of failures. Mutation operators in this category delete or alter lifecycle methods of activities and services, such as `onPause`, `onResume`, or `onDestroy`. These mutations simulate faults where developers fail to properly save or restore state, or incorrectly manage long-running services [9, 21]. Lifecycle-related operators are particularly relevant for testing background and foreground transitions.

XML- and resource-based mutation operators target the non-code artifacts that define much of an Android application's behavior. Layout files, manifests, and resource declarations are critical

for correct execution, yet they are not exercised by traditional code-level mutation operators. Several approaches define operators that delete or modify GUI widgets, change layout attributes, or remove permissions from the manifest file [9, 19]. Examples include deleting buttons or text fields from layouts, switching widget positions, or removing required permissions. These operators aim to expose inadequacies in GUI testing and configuration validation.

Context-aware and non-functional mutation operators address aspects that go beyond functional correctness. Android applications often depend on external context, such as device location, orientation, network connectivity, or energy usage. Mutation operators in this category modify context-dependent inputs or system interactions. For instance, location modification operators alter latitude or longitude values to simulate unexpected context changes, while orientation-lock operators force the application into a fixed screen orientation [9]. Other operators target energy-related behavior by deleting wake-lock release calls, modeling energy leaks [33].

Several tools implement subsets of these operators with different scopes and emphases. muDroid implements a broad set of Android-specific operators and combines them with traditional Java operators, targeting both functional and non-functional faults [33]. Edroid focuses primarily on GUI and XML-based operators, mutating layout and manifest files to assess test suites that rely on black-box or GUI-level testing [19]. More recent tools such as MDroid+ and DroidMutator expand operator catalogs and introduce mechanisms to reduce invalid or redundant mutants, for example by applying type checking or restricting mutation scope [18, 20].

Overall, the literature shows a clear shift toward mutation operators that better reflect how Android applications actually fail. Instead of relying only on traditional statement-level mutations, more recent work focuses on faults that have been observed in real applications. Many mutation operators are derived from empirical studies of bug reports and development histories, and target issues such as incorrect component interaction, lifecycle misuse, configuration mistakes, and context-dependent behavior. This shift aims to make mutation testing more representative of real faults and more useful in practice.

3.2 Mutation Testing at Assembly-Level

Most mutation testing approaches for Android operate at source-code level and assume that the application source code is available. While this assumption is reasonable for open-source projects and some development settings, it does not hold in many practical scenarios. In industrial contexts, Android applications are frequently tested by third-party services, outsourced teams, or cloud-based infrastructures that only receive the compiled APK. In such cases, source-code based mutation testing cannot be applied.

Assembly-level mutation testing addresses this limitation by applying mutations directly to the compiled representation of the application. Instead of modifying Java or Kotlin source code, mutations are introduced at APK level, using intermediate representations extracted from the package. This makes mutation testing applicable to both open- and closed-source Android applications, and better reflects how apps are distributed and tested in practice [10, 11].

Another reason for working at assembly level is independence from the original programming language. Android applications can be implemented using different languages and frameworks, but they all compile to the same bytecode format. By operating at APK level, mutation testing becomes language-agnostic and avoids relying on language-specific tooling or build pipelines, which are often unavailable or difficult to reproduce outside the original development environment.

3.2.1 Existing Implementations

Assembly-level mutation testing relies on intermediate representations extracted from APK files. Among the available representations, smali has become the most commonly used target. The main reason for this choice is that smali supports reliable disassembly and reassembly of APKs. In contrast, decompiling Android applications to Java source code often produces code that cannot be compiled again, due to lost information or imprecise reconstruction. Empirical studies show that smali-based transformations allow applications to be disassembled and reassembled into executable APKs with a high success rate, making mutation testing feasible at this level [11]. This property is essential for mutation testing, since each mutant must result in an installable and runnable application.

In existing implementations, the mutation process typically begins by unpacking the APK and decoding its bytecode and resources into smali and XML files. Potential mutation locations are then identified using static analysis. In the case of MutAPK, this analysis is guided by a Potential Fault Profile, which associates code and resource locations with applicable mutation operators based on predefined fault patterns [12]. Mutants are generated by creating a modified copy of the decoded application for each selected location and applying exactly one mutation per copy.

After mutation, the modified artifacts are reassembled into APK files and signed to ensure they can be installed on Android devices or emulators. This signing step is required because the original application signature is lost during repackaging. The output of the process is a set of APK-level mutants, each corresponding to a single injected fault and ready for execution.

MutAPK follows this workflow and focuses primarily on mutant generation, producing APK mutants without executing test suites as part of the tool itself [11]. MuDroid applies a similar smali-based mutation approach, but integrates mutant execution by running the mutated APKs under automated interaction testing and observing behavioral differences during execution [33]. In both cases, mutation is performed directly on the executable representation of the application, without reconstructing or relying on high-level source code.

3.3 Cost Reduction and Optimization Techniques

Mutation testing is a well-known technique for assessing the quality of test suites, but it is also widely recognized as being computationally expensive. The cost mainly comes from generating a large number of mutants, compiling and deploying them, and executing the test suite for each mutant. In the context of Android applications, this cost is even higher, since mutation testing often operates at the APK level, requiring applications to be rebuilt and installed on emulators or

physical devices. Several studies point to execution overhead as one of the main obstacles to the practical adoption of mutation testing for Android apps [6].

To deal with this problem, the literature proposes a variety of cost reduction techniques. These techniques are usually grouped into a few broad strategies, such as reducing the number of mutants, avoiding mutants that are not useful, and optimizing how mutants are executed. Empirical studies and literature reviews show that effective mutation testing tools typically combine multiple strategies, rather than relying on a single optimization.

3.3.1 Reducing the Number of Mutants

Reducing the number of generated mutants is one of the most common ways to lower the cost of mutation testing. Running mutation testing exhaustively is rarely practical for real-world systems, especially for Android applications of non-trivial size. To address this, many approaches apply mutant selection techniques that aim to keep a representative set of faults while working with fewer mutants.

Random selection is the simplest approach, but more refined techniques use statistical sampling or take mutation operators into account to ensure that different types of faults are still covered. Empirical results from Android mutation testing show that these techniques can significantly reduce the number of executed mutants while producing mutation scores that are close to those obtained using the full mutant set [13, 33].

3.3.2 Avoiding Useless Mutants

Another source of unnecessary cost comes from mutants that do not contribute meaningful information. This includes dead-code mutants, duplicate mutants, and equivalent mutants. In Android applications, dead code is relatively common in APKs, since the default build process does not remove unreachable methods. Mutating such code leads to mutants that are never executed and therefore cannot be killed by any test.

Static analysis techniques, such as building a call graph, can be used to identify unreachable parts of the code and exclude them from mutation. Studies show that removing dead-code mutants can considerably reduce the total number of generated mutants without negatively affecting the mutation score [13]. Duplicate and equivalent mutants introduce additional redundancy. Although detecting semantic equivalence is undecidable in general, practical tools often rely on simple syntactic checks, such as hashing mutated artifacts, to detect mutants that are identical to the original program or to each other. While approximate, these techniques are cheap to apply and work well in practice.

3.3.3 Optimizing Mutant Execution

Even after reducing the number of mutants, executing them remains one of the main cost factors in Android mutation testing. In traditional approaches, each mutant is packaged as a separate APK, which means that the application has to be rebuilt, installed, and initialized before running the

tests. On Android, these steps are particularly expensive, especially when tests are executed on emulators or physical devices.

Execution-level optimizations focus on reducing this overhead without further limiting the set of mutants. A key technique in this area is mutant schemata, where multiple mutants are embedded into a single application and selected at runtime. Instead of generating one APK per mutant, a single instrumented APK contains all mutations, and a runtime parameter determines which mutant is active. This allows all mutants to share the same compilation, installation, and initialization steps.

This approach has several advantages. It greatly reduces disk usage, since only one mutated application needs to be stored. It also avoids repeated build and deployment steps, which are a major source of overhead in Android mutation testing. By eliminating repeated application startup, mutant schemata usually lead to noticeable execution-time improvements compared to the traditional one-mutant-per-APK approach. Empirical studies report that mutant schemata preserve the same behavior as traditional mutation testing while achieving these efficiency gains [28, 32].

At the same time, mutant schemata have important limitations. The technique relies on control-flow constructs, such as conditional statements or switch blocks, to select the active mutant. In Java, these constructs can only be placed inside method bodies. As a result, mutations that affect elements outside method scope cannot be directly encoded using mutant schemata. This includes mutations on class-level fields, static initializers, method signatures, annotations, and some configuration or resource-related elements. Such mutants require special handling or are excluded from schemata-based execution.

Some mutation operators are also harder to support when multiple mutants are embedded in the same method. Operators that modify control flow or variable declarations may require additional code transformations, such as splitting declarations or introducing temporary variables. This increases implementation complexity and can limit the set of supported operators. Moreover, embedding many mutants in a single application increases code size and control-flow complexity, which may introduce small runtime overheads during mutant selection.

Other execution optimizations focus on the test execution process itself. Early stopping techniques terminate test execution for a mutant as soon as it is killed, avoiding unnecessary test runs. Reordering test cases can further improve this effect by increasing the chances of killing mutants early. Finally, reducing the number of emulator restarts can significantly decrease execution time, since emulator startup and state initialization often dominate the overall cost in Android testing environments [6].

3.4 Obstacles and Open Challenges

Despite the advances made in mutation testing for Android applications, several challenges remain. One recurring issue is the large number of mutants produced by Android-specific operator sets. Many of these mutants are redundant, equivalent, or invalid, which increases both computational cost and manual effort. Although different techniques have been proposed to reduce this

overhead, equivalent mutant detection remains an open problem and is still handled in a limited and approximate way.

Another challenge relates to the observability of mutant behavior. Many Android testing approaches rely on crashes, exceptions, or simple output comparison as test oracles. These oracles are often insufficient to detect behavioral differences introduced by mutants, especially when changes affect the user interface, component interaction, or internal state without causing a failure. This limits the ability of mutation testing to fully assess test effectiveness in realistic Android scenarios.

The execution model of Android applications further complicates mutation testing. Android apps are event-driven and interact with a managed runtime that controls component lifecycles, scheduling, and resource allocation. As a result, application behavior can vary depending on execution order, device state, or external context, making mutant execution costly and sometimes difficult to reproduce consistently.

Finally, tool support remains fragmented. Many Android mutation testing tools are designed with a narrow focus, such as a specific class of mutation operators or a particular execution strategy. This specialization limits their general applicability and makes it difficult to reuse tools across different contexts or to compare results obtained by different approaches.

3.5 Literature Review

To address the research questions and explore existing solutions, an exploratory literature review was conducted.

3.5.1 Search

The first 2 papers were recommended by the advisors, which work as a base to the research that follows.

The initial search was made using IEEE XPlore, Google Scholar and Elicit. Below are the queries used for each source. Note that for Elicit, since it uses Artificial Intelligence, the search used natural language instead of query operators. The queries used are presented in Table 3.1.

Database	**Search Query**
IEEE Xplore	("Android" OR "mobile app" OR "APK" OR "smali") AND ("mutation operator" OR "mutation testing" OR "mutation analysis")
Google Scholar	("Android" OR "mobile app" OR "APK" OR "smali") AND ("mutation operator" OR "mutation testing" OR "mutation analysis")
Elicit	Mutation testing on Android applications

Table 3.1: Search queries used for literature review

The searches retrieved 87 papers, which were imported to Zotero for screening.

3.5.2 Selection

After retrieving the base results, many of the papers were either not related or not relevant to the research, so a filter step was performed.

We analyzed the title and the abstract to select which papers were relevant. The main themes that were searched were mutation testing strategies and mutation testing focused on Android, assembly or not. Many papers were excluded because, even though they talked about mutation testing, they were more focused on a specific language or platform, which were not related to this work.

From the total of 89 papers, 23 were selected to study for the state of the art.

To retrieve even more relevant information, we performed snowballing on two papers considered very relevant to the work [11] [13], and found 6 that matched the selection criteria, bringing the total research papers number to 29.

3.5.3 Selected Papers

The complete list of selected papers is provided in the bibliography.

Chapter 4

metford-apk: Assembly-Level Schemata Mutation Testing for Android

This chapter describes the design and implementation of metford-apk, a mutation testing tool that operates directly on the Dalvik bytecode of Android applications. Rather than requiring access to Java or Kotlin source code, the tool works from a compiled APK, making it applicable to any Android application regardless of how it was built or whether its source is available. The tool implements the mutant schemata strategy at the assembly level and provides a set of fourteen mutation operators that mirror those of the METFORD source-level tool [32].

4.1 Overview

Applying the mutant schemata strategy to Android assembly requires working within a different set of constraints than at the source level. At the source level, mutants are introduced as alternative branches in a Java `switch` statement, which the compiler then lowers to bytecode alongside the rest of the code. At the assembly level, there is no compiler to help: the schemata block must be constructed directly in Dalvik bytecode, inserted into an existing method, and kept consistent with all surrounding instructions. The register model of Dalvik adds a further constraint that does not exist at the source level: each method declares a fixed number of registers upfront, and introducing a schemata block may require an extra register that pushes existing parameter registers beyond the range that certain instruction formats can encode. These challenges are specific to the assembly representation and must be addressed independently of which particular operators are being applied.

The approach proposed in this work is to perform mutation testing directly on the compiled APK, without access to source code. The APK is decoded to its smali assembly representation, mutation operators are applied directly to the smali instructions, and the result is repackaged into a new APK. Using the schemata strategy at this level means that all mutants are embedded in a single APK, so the application needs to be installed only once and the active mutant is selected

at runtime. This makes bytecode-level mutation testing practical at the same scale as source-level schemata approaches.

The starting point for this work was METFORD [32], a source-level mutation testing tool for Android that defines a catalog of Android-specific mutation operators and applies them using the mutant schemata strategy. METFORD operates on Java source code and relies on a full build of the application to produce a mutated APK. This dependency on source code limits its applicability: many Android applications are distributed only as APKs, and the build environment required to recompile a large project is not always available or reproducible. Moving mutation to the assembly level lifts this restriction and makes the approach applicable to any Android APK regardless of its origin.

A key decision was to mirror the operator set of METFORD as closely as possible. Since one of the goals of this work is to compare source-level and bytecode-level mutation testing, having both target the same classes of faults is essential for a meaningful comparison. The fourteen operators cover Android-specific fault categories including GUI interaction, intent communication, view handling, and arithmetic operations. Adapting them to the assembly level required understanding how each source-level construct maps to Dalvik instructions, which was not always straightforward: some patterns that are simple to identify in Java source are spread across multiple instructions in smali, while others that are implicit at the source level become explicit in bytecode.

A further challenge is that bytecode-level analysis has no access to the Android framework class hierarchy. An APK contains only the application's own classes; the framework classes (`Activity`, `Fragment`, `View`, and so on) reside in the Android runtime on the device and are not part of the APK itself. At the source level, a tool can use the Java type system to restrict an operator to call sites that appear inside subclasses of a specific base class, for example limiting a `findViewById` operator to methods declared in `Activity` subclasses. At the bytecode level, following the inheritance chain upward requires resolving framework classes that are simply not present, so this kind of type-aware scoping is not directly available. As a result, operators that in a source-level tool are scoped to specific class hierarchies must instead match on call signatures alone, which can produce a broader set of mutation sites than intended.

metford-apk is a prototype that implements this approach. It takes an Android application and a configuration file as input, decodes the application, identifies locations where each operator applies, and injects all mutants using schemata encoding. The result is a single APK containing every mutant alongside the original code, together with a mutation report recording the location and nature of each mutant.

Figure 4.1 illustrates the overall pipeline.

4.2 Mutant Schemata at the smali Level

The mutant schemata strategy embeds all mutants into a single program and uses a runtime switch to select which mutant executes during a given test run [28]. In METFORD, this is implemented at the Java source level as a `switch` statement around each mutation site. Adapting the same

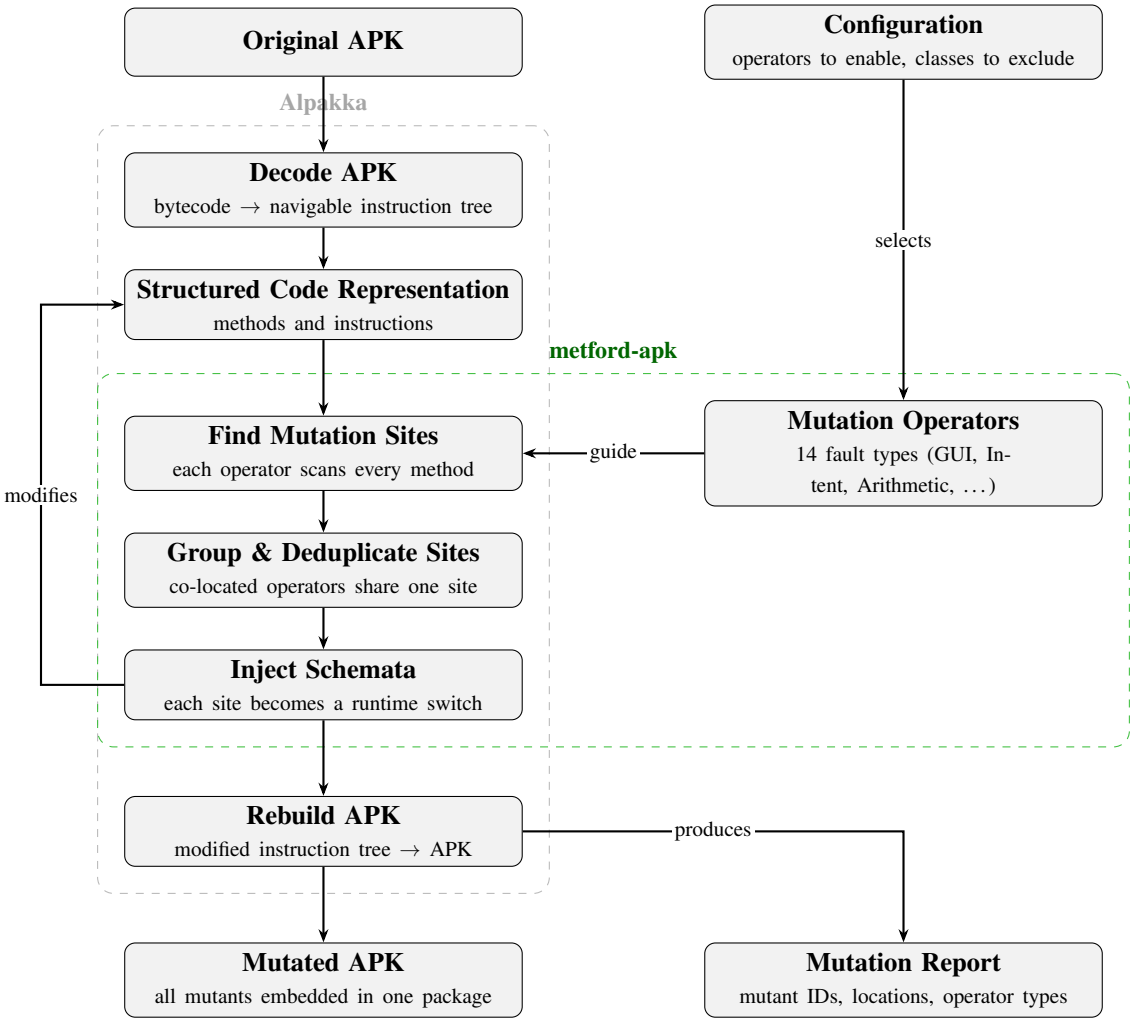


Figure 4.1: metford-apk pipeline overview. An input APK is decoded, mutated by the configured operators using schemata injection, and rebuilt into a single APK containing all mutants.

strategy to smali follows the same idea, but requires working within the constraints of the Dalvik register model in particular, injecting a schemata block may require an extra register, which is not always possible without invalidating the method's bytecode. This is addressed by a per-method safety check described in Section 4.3.

At the smali level, Dalvik provides two switch instructions. The `sparse-switch` maps arbitrary integer keys to branch targets and is suitable when the case values are spread out. The `packed-switch` requires consecutive case values starting from a base, but is more compact and efficient when the range is dense. Since mutant IDs are assigned sequentially, all cases at a given site form a consecutive range, making `packed-switch` the natural choice.

To implement schemata at the smali level, each mutation site is replaced by a dispatch block. The block starts by reading a global integer field, `MUTANT_ID`, which controls which mutant is active during a given test run. Based on its value, the block jumps to one of several code variants, each containing the mutated version of the original instructions. If the value does not match any variant, the original code runs unchanged.

Each variant is a separate labeled block in the method body. The `packed-switch` instruction maps the value of `MUTANT_ID` to the corresponding label. Since mutant IDs are assigned consecutively, the cases always form a dense range, which is exactly what `packed-switch` requires. A separate data table at the end of the block lists the target labels for each case.

When no case matches, execution falls through past the `packed-switch` and reaches the original code block. Each variant and the original block end with a jump to a shared exit label, so only one path executes per site. Listing 4.1 shows the full structure for a site with two variants.

Listing 4.1: Schemata block for a site with two variants.

```
sget v_tmp, LMutationController;->MUTANT_ID:I
packed-switch v_tmp, :pswitch_data
goto :default          # fall-through when no case matches

:case_0
    <variant 0 code>
    goto :end
:case_1
    <variant 1 code>
    goto :end
:default
    <original code>
    goto :end

:pswitch_data
.packed-switch 0x<base_id>
    :case_0
```

```
    :case_1
.end packed-switch
:end
```

The switch is controlled by a mutant identifier that is set externally before each test run. When no identifier is set, the default block executes and the application behaves as the original. How this identifier is injected into the APK and activated at runtime is described in Section 4.4.

As seen in Listing 4.1, the schemata block uses a temporary register, `v_tmp`, to hold the value of `MUTANT_ID` before the switch. This register does not exist in the original method and must be added by incrementing the method's local register count by one. Most operators require this extra register, since they all need to read `MUTANT_ID` at the start of the block. A small number of operators that only modify the method without needing a new register can be applied without expanding the frame. Whether it is safe to add a register depends on the existing register layout of each method, and is checked before any operator is applied. This is described in detail in Section 4.3.

When two operators identify the same instruction as a mutation point, their variants are merged into a single `packed-switch` block rather than producing two separate sites. This reduces the number of switches in the method and avoids conflicting smali modifications at the same anchor point.

4.3 Register Safety

Unlike Java, where the compiler allocates registers freely, smali methods work with a fixed register frame declared upfront. Each method declares n local registers (`v0` through $\mathtt{v}_{n-1}$) followed by p parameter registers ($\mathtt{v}_n$ through $\mathtt{v}_{n+p-1}$). Injecting a schemata block requires a temporary register to hold the active mutant ID, which means expanding this frame by one. Doing so shifts all parameter registers upward by one slot.

The problem is that certain Dalvik instruction formats can only encode register numbers up to 15. In particular, the instructions used to call methods in smali, the `invoke-*` family, exist in multiple encoding formats. The most common format, used for calls with up to five arguments, encodes each register number in four bits, which limits it to values between 0 and 15. A method with many locals and parameters may already place some parameter registers at or near this boundary. Adding one extra local pushes them over it, producing bytecode that the Dalvik verifier rejects at install time.

One approach to prevent this is to compute a safety margin for each method before applying any operators. The margin represents how many extra local registers can be added without pushing any parameter register above the encodable limit:

$$safeExtraSlots = \max(0,\ 15 - locals - numParams + 1)$$

Where *locals* is the current `.locals` count and *numParams* is the number of register slots occupied by parameters. Wide types (`J` for long, `D` for double) each occupy two slots and are counted accordingly. Operators that require more extra registers than *safeExtraSlots* are skipped for that method. This is a conservative check that prevents any invalid bytecode from being produced, at the cost of leaving some mutation sites in register-constrained methods uncovered.

4.4 Mutation Controller

For the schemata dispatch to work at runtime, the mutant identifier must be accessible from anywhere in the application. metford-apk injects a helper class, `MutationController`, directly into the APK. This class contains a single static integer field, `MUTANT_ID`, which is read by every schemata block.

The class is not part of the original application: it is generated as a smali file and placed in the APK alongside the application code during the build step. Because it is a static field on a class that is loaded once per process, it is visible to all methods that reference it, regardless of which class or package they belong to.

The value of `MUTANT_ID` is set externally before each test run via an Android system property. The application reads the property at startup and stores its value in the field. Activating a specific mutant requires setting the property and restarting the application so the new value takes effect:

```
adb shell setprop debug.mutant.id 5
adb shell am force-stop <package>
adb shell am start <package>/<activity>
```

When the property is absent or set to `-1`, the field holds a value that does not match any case in any schemata block, so all dispatch blocks fall through to the original code and the application behaves as the unmodified version.

Each mutation site is assigned a range of consecutive integer IDs. A site with k variants occupies IDs $[base,\ base + k - 1]$, where *base* is drawn from a global counter that increments across all methods and sites. This ensures that every mutant across the entire APK has a unique identifier.

4.5 Contributions to the Alpakka Framework

Development of metford-apk required extending Alpakka in two areas where the framework lacked the necessary abstractions: a structured representation of binary operations and a factory for constructing packed-switch control structures.

4.5.1 Binary Operator Hierarchy

Alpakka originally represented every instruction solely through its opcode string. There was no distinction between the base operation name, the operand type, and the instruction format modifier. For binary operations this distinction matters: the opcode `add-int/lit8` encodes three independent pieces of information: the base operator (`add`), the operand type (`int`), and the format modifier (`/lit8`), and a mutation that replaces the operator must reconstruct a valid opcode that preserves the other two components.

To address this, a `BinaryOperator` enumeration was added to Alpakka that classifies all binary operators into arithmetic operators (`add`, `sub`, `mul`, `div`, `rem`, `rsub`) and bitwise operators (`and`, `or`, `xor`, `shl`, `shr`, `ushr`). A companion `OperandType` enumeration captures the numeric domain of the operands (`int`, `long`, `float`, `double`). These are combined in a `BinaryOp` abstract class that covers all Dalvik binary instruction formats (12x, 23x, 22b, 22s). The class exposes a typed `setOperator()` action that validates that arithmetic and bitwise operators are not mixed, and reconstructs the opcode by preserving the operand type suffix and any format modifier. A `fromName()` helper strips the type suffix from a raw opcode string to recover the base operator name. The Alpakka weaver exposes these as a `binaryOp` join point with an `operator` attribute and a `setOperator()` action. An Alpakka script can therefore write `binaryOp.setOperator("sub")` and receive back the correct Dalvik opcode for any operand type and format combination, without manually parsing or reconstructing opcode strings.

4.5.2 Packed-Switch Construction

Alpakka provides operations for inserting and replacing individual statements as literal strings, but had no facility for constructing a complete packed-switch control structure as a set of structured AST nodes. A `packedSwitch()` factory method was added to `SmaliFactory` that accepts a register name and a list of `PackedSwitchCase` objects. Each `PackedSwitchCase` groups a case entry label with its body statements into a single AST node. The factory method assembles the full structure: the `packed-switch` branch instruction that references the data block, the sequence of case body blocks each preceded by its label, and the `.packed-switch` data directive that enumerates the case targets. This provides a structured API for generating switch dispatch code from within Alpakka scripts.

In metford-apk, the schemata switch is constructed as a formatted text block and inserted into the method body using Alpakka's `insertBefore` operation. The `SmaliFactory` addition enables the same construction through the AST API, which can serve as the basis for future tools that need to generate or transform switch-based control flow at the AST level rather than at the text level.

4.6 Mutation Operators

metford-apk implements fourteen operators drawn from the METFORD operator set [32], covering traditional fault models and Android-specific fault patterns. The operators are grouped into three categories: traditional, Intent, and GUI.

Table 4.1 summarises the fourteen operators, their category, and the fault they simulate.

Table 4.1: Mutation operators implemented in metford-apk, based on the METFORD operator set [32].

Category	**Operator**	**Fault simulated**
Traditional	ArithmeticOperatorMutator	Replaces arithmetic operators (`+`, `-`, `*`, `/`) with alternatives
Intent	NullIntentMutator	Replaces the Intent passed to `startActivity`/`startService` with null
Intent	IntentTargetReplacementMutator	Redirects an explicit Intent to a different component in the same package
Intent	NullPutExtraValueMutator	Replaces the value in `putExtra` with null
Intent	NullPutExtraKeyMutator	Replaces the key in `putExtra` with null
Intent	InvalidKeyIntentMutator	Replaces the Context argument of an Intent constructor with null
Intent	RandomIntentActionMutator	Replaces the Intent constructor with one using `ACTION_VIEW` or `ACTION_SEND`
GUI	BuggyGUIListenerMutator	Nullifies the listener in `setOnClickListener`
GUI	LengthyGUIListenerMutator	Injects a 10-second sleep in event listener callbacks
GUI	LengthyGUICreationMutator	Injects a 10-second sleep after `super.onCreate`
GUI	FindViewByIdReturnsNullMutator	Replaces the result of `findViewById` with null
GUI	InvalidIDFindViewMutator	Replaces the ID argument of `findViewById` with an invalid value
GUI	InvalidViewFocusMutator	Appends a `requestFocus` call on the returned view
GUI	ViewComponentNotVisibleMutator	Appends a `setVisibility(INVISIBLE)` call on the returned view

Adapting mutation operators from Java source to smali requires understanding how high-level constructs map to bytecode, and this translation is not always one-to-one.

In Java, arithmetic operations are expressions that can appear inside larger expressions and be nested arbitrarily. A statement like `x = a + b * c` is a tree of operations that the compiler resolves during compilation. At the smali level, this tree has already been flattened: each operation is a separate instruction that reads from and writes to explicit registers, in a linear sequence. Rather than traversing a nested expression tree, the operator only needs to match individual instructions

such as `add-int` or `mul-float`. The challenge is that Dalvik defines many variants of each operation, one per operand type and instruction format, all of which must be handled correctly.

For other operators, the construct of interest is spread across multiple instructions. A single Java method call, for example, compiles into an `invoke-*` instruction followed by a `move-result-object` to capture the return value, and sometimes additional instructions before or after. Each operator must match the right sequence and produce a valid replacement without disturbing the surrounding code.

4.6.1 Traditional Operators

ArithmeticOperatorMutator: Replaces arithmetic binary operators with configured alternatives. The four configured substitutions are `add`→`sub`, `sub`→`add`, `mul`→`div`, and `div`→`mul`. At the source level this is a simple AST node replacement. In smali, arithmetic operations exist in multiple encoding forms depending on the operand type and instruction format: integer, long, float, and double variants, as well as literal-operand forms (`/lit8`, `/lit16`). The operator must identify the base operation, preserve the operand type and format modifier, and reconstruct a valid opcode for the replacement. An additional Dalvik-specific case arises because there is no `sub-int/lit8` opcode: for `add`→`sub` mutations on literal forms the literal operand is negated instead. For example, `add-int/lit8 v0, v1, 5` becomes `add-int/lit8 v0, v1, -5`, which is arithmetically equivalent to subtracting 5.

4.6.2 Intent Operators

Android applications communicate between components using `Intent` objects. The Intent operators simulate faults in this communication mechanism. In Java source, intent operations are visible as method calls on well-typed objects. In smali, the same operations appear as `invoke-virtual` or `invoke-direct` instructions with fully qualified method signatures as string constants. The operators match on these signature strings to identify the relevant call sites.

NullIntentMutator: Replaces the `Intent` object passed to `startActivity` or `startService` with null. The operator identifies the `new-instance` and constructor call sequence that creates the Intent object and replaces the entire sequence with a null assignment, so the subsequent launch call receives a null intent.

IntentTargetReplacementMutator: Replaces the target class in an explicit Intent constructor (`new Intent(context, TargetActivity.class)`) with an alternative class from the same package. The operator locates the `const-class` instruction that loads the target class descriptor and replaces it with the descriptor of the first alternative class found in the same package. This simulates a routing fault where the wrong component is launched.

NullPutExtraValueMutator: Replaces the value argument of `Intent.putExtra` calls with null, while keeping the key. This simulates data-passing faults where an extra is present but empty.

NullPutExtraKeyMutator: Replaces the key argument of `Intent.putExtra` calls with null. An intent extra registered under a null key cannot be retrieved by the receiving component.

InvalidKeyIntentMutator: Replaces the Context argument of explicit Intent constructors with null. This simulates a missing context fault that causes the intent to be constructed without a valid package reference.

RandomIntentActionMutator: Replaces the entire Intent construction sequence with a new Intent created using a fixed action string. In METFORD, one of two possible actions (`ACTION_VIEW` or `ACTION_SEND`) is chosen randomly per site using a seed, avoiding the original value. In metford-apk, both are produced as separate variants, which achieves the same effect under schemata: at least one variant is always different from the original.

4.6.3 GUI Operators

The GUI operators simulate faults in view management and user interaction handling. A recurring smali-level challenge in this group is that the result of `View.findViewById` the primary way views are retrieved in Android is followed by a `move-result-object` instruction that stores the returned view into a register, and may then be immediately used as the receiver of a chained method call. The operators in this group must distinguish between sites where the result is stored for later use and sites where it is immediately chained, since the mutation target differs between the two cases.

BuggyGUIListenerMutator: Nullifies the listener argument in `View.setOnClickListener` calls, effectively removing the listener registration. This simulates a copy-paste fault where a listener is attached to the wrong view or not attached at all.

LengthyGUIListenerMutator: Injects a `Thread.sleep(10000)` call at the entry point of event listener callbacks such as `onClick` and `onTouch`. Blocking the UI thread for this duration simulates an ANR fault caused by a long-running operation in a listener. The 10-second value is inherited from the original METFORD operator. In smali, injecting a sleep call requires a wide register pair to hold the long argument, which adds an additional register constraint on top of the schemata register requirement.

LengthyGUICreationMutator: Injects a `Thread.sleep(10000)` call after the `super.onCreate` invocation in activity and fragment creation methods. Blocking during creation simulates slow initialisation that causes the application to miss Android's activity startup deadline. The 10-second value is inherited from the original METFORD operator.

FindViewByIdReturnsNullMutator: Replaces the result of `View.findViewById` calls with null. After the call, Android code typically stores the result in a register via `move-result-object`; the operator replaces this instruction with a null assignment to the same register. Sites where the result is immediately used as the receiver of a chained method call are excluded, since nullifying the result there would produce a different fault than intended.

InvalidIDFindViewMutator: Replaces the integer ID argument of `View.findViewById` with an invalid value immediately before the call. A zero resource ID is not a valid view identifier, so the call returns null and any subsequent dereference of the result produces a `NullPointerException`. In smali, injecting the invalid constant requires an extra `const` instruction before the `invoke-virtual`, which must be inserted without disturbing the surrounding register assignments.

InvalidViewFocusMutator: Appends a `View.requestFocus` call on the view returned by `findViewById`, simulating a focus management error where an unintended view receives focus.

ViewComponentNotVisibleMutator: Appends a `View.setVisibility` call after `findViewById`, passing `INVISIBLE` (0x4) as the visibility constant. This hides the view from the user while keeping it in the layout hierarchy. The call requires a temporary register to hold the constant, adding to the register pressure on the method frame.

4.7 Configuration and Scope Filtering

Each subject application has a dedicated JSON configuration file. The configuration specifies the input APK, the output APK and report paths, and the list of operators to apply. Arithmetic operators take a `from`/`to` argument pair; all other operators are listed by name.

Two filtering fields control the scope of the analysis. The `packageFilter` field limits Alpakka to loading only smali files under a given path prefix, restricting the analysis to the application's own classes and excluding third-party libraries. Without this filter, Alpakka would load all smali files in the APK, which in large apps can reach tens of thousands of methods and make the weaver impractically slow. The `excludeClasses` field accepts a list of substrings matched against method descriptors; methods whose descriptor contains any listed substring are skipped entirely. This mirrors the `excludeList` mechanism in METFORD, and the same exclusion lists from the original METFORD paper configurations are applied in metford-apk to ensure a consistent comparison.

4.8 Mutation Report

After each run, metford-apk writes a JSON report containing all mutation sites found in the application. For each site the report records the method descriptor, the smali line number, the original

instruction code, and the list of variants produced at that site. Each variant entry includes the name of the operator that generated it and the mutant smali code.

The line number is derived from the nearest `.line` directive preceding the mutation anchor in the smali stream. Dalvik compilers emit one `.line` directive per Java source line, covering all smali instructions that originate from that line. A Java statement that expands to several smali instructions will therefore have a directive only at the first of those instructions; if the mutation anchor falls on a later instruction from the same source line, no directive precedes it and the line number is recorded as null. In practice, a significant number of sites produce null line numbers for this reason, either because the anchor lands between directives or because the line information was not correctly surfaced through the Alpakka join-point model. The consequences of this limitation for cross-tool comparison are discussed in Section 5.7.

Chapter 5

Evaluation

This chapter evaluates the approach proposed in this work against the research questions stated in the introduction. RQ1, which asked how source-level mutation strategies can be applied at the assembly level, is addressed by the design and implementation described in Chapter 4. The evaluation here addresses RQ2 and RQ3.

RQ2 asked which source-level operators remain meaningful at the assembly level. This is evaluated by comparing the mutant counts and mutation sites produced by the assembly-level prototype against those of METFORD, examining where the two approaches agree and where they diverge. RQ3 asked whether assembly-level mutation improves efficiency and applicability compared to source-level. This is evaluated through build time, schemata APK size, mutation scores, and the ability to process closed-source applications.

The comparison is carried out between metford-apk and METFORD across a set of Android applications.

5.1 Subject Applications

The evaluation uses nine Android applications. Six are open-source and allow a direct comparison between METFORD and metford-apk, since both tools require access to the same application in different forms: METFORD from Java source, metford-apk from a compiled APK. Three additional closed-source applications are used to evaluate metford-apk alone, demonstrating the tool's ability to operate without source code access (Section 5.6).

The subject applications come from three sources. The six open-source applications are the same ones used in the original METFORD study [32]. CleanCalculator comes from the MUDROID dataset [33]. CPU-Z and Musicolet are closed-source applications obtained as released APKs: CPU-Z is a small system-information tool, and Musicolet is a widely used offline music player, included to show that metford-apk works on arbitrary APKs without source code and scales to a substantial commercial application.

Table 5.1 summarises the subject applications. APK sizes refer to the original, unmodified release. Musicolet is distributed as a split application bundle; the size shown, and all later measurements, use a single universal APK merged from its released splits.

Table 5.1: Subject applications.

Application	Category	Source	APK size
AmazeFileManager	File manager	Open	35.6 MB
AntennaPod	Podcast player	Open	17.1 MB
Aegis	Two-factor authenticator	Open	14.0 MB
KeePassDroid	Password manager	Open	11.0 MB
Omni-Notes	Note-taking	Open	28.8 MB
Simplenote	Note-taking	Open	13.0 MB
CleanCalculator	Calculator	Closed	0.7 MB
CPU-Z	System information	Closed	5.1 MB
Musicolet	Music player	Closed	32.1 MB

For the open-source subjects, the same exclusion lists used in the original METFORD paper are applied to both tools, ensuring that the comparison is not skewed by scope differences at the file level.

5.2 Mutation Count Comparison

RQ2: Which source-level mutation operators remain meaningful when applied at the assembly level?

Table 5.2 shows the total number of mutants generated by each tool across the six open-source subject applications. metford-apk produces more mutants than METFORD in all cases. The difference ranges from 51 additional mutants on Aegis to 1240 on AntennaPod.

Table 5.2: Total mutant counts per application.

Application	METFORD	metford-apk	diff
AmazeFileManager	1040	1632	+592
AntennaPod	1216	2456	+1240
Aegis	979	1030	+51
KeePassDroid	406	566	+160
Omni-Notes	321	501	+180
Simplenote	804	1474	+670

The overall difference is not uniform across operators. The full per-operator counts for every subject are given in Appendix A; the per-operator observations below summarise the main differences.

`FindViewByIdReturnsNullMutator` and `InvalidIDFindViewMutator` produce substantially more mutants in metford-apk in every subject. METFORD can restrict these operators to

call sites inside subclasses of specific Android UI base classes, whereas metford-apk has no access to the framework class hierarchy and must match on call signatures alone, applying the operators to any `View.findViewById` call site regardless of the enclosing class. Whether the additional sites represent meaningful fault locations or noise is an open question.

`ArithmeticOperatorMutator` is also generally higher in metford-apk, markedly so on the larger subjects (for example AmazeFileManager and AntennaPod roughly double). The main cause is that the Java compiler emits arithmetic instructions for constructs that contain no explicit arithmetic operator in the source. A for-each loop over a collection, for instance, compiles to an iterator with an internal counter; enhanced for loops over arrays compile to an index variable that is incremented on every iteration. These compiler-generated instructions are invisible to a source-level tool but are present in the bytecode and are matched as valid arithmetic mutation sites. On a few subjects the count is instead lower (for example Aegis), where some arithmetic sites fall in methods that the register-safety constraint excludes.

`LengthyGUIListenerMutator` registers zero or very few mutants in METFORD while metford-apk finds a significant number, as the METFORD configurations exclude several activity classes that contain listener registrations. `IntentTargetReplacementMutator` reports zero for METFORD across all subjects; this operator requires that an alternative class be available in the same package at the time of analysis, which is more reliably satisfied when inspecting the compiled APK than when analysing source code.

Some intent operators (such as `RandomIntentActionMutator` and the `NullPutExtra` pair) instead produce fewer mutants in metford-apk, because its bytecode pattern matching is conservative: it mutates an intent construction or `putExtra` call only when the surrounding instructions match the exact pattern it expects, and skips the rest. A concrete example is `Intent.putExtra`, which has a distinct overload for each value type `String`, `int`, `boolean`, `Parcelable`, and others. In smali, each overload is a separate fully qualified method signature, and metford-apk matches on the full signature including parameter types rather than on the class and method name alone. An operator that covers only a subset of overloads therefore silently skips any call using a different one. At the source level, METFORD sees `intent.putExtra(key, value)` as a single AST node regardless of the value type and matches it unconditionally. On Simplenote, for instance, metford-apk matches only about 40% of the intent-construction and `putExtra` sites present in the bytecode. Relaxing the signature matching to cover all overloads is a straightforward improvement left for future work.

5.3 Mutation Site Correspondence

RQ2: Which source-level mutation operators remain meaningful when applied at the assembly level?

A direct line-level comparison between the two tools is not feasible. As discussed in Section 5.7, a significant number of metford-apk mutation sites carry a null line number due to missing `.line` directives in the smali code. The comparison is therefore performed at the class level:

a METFORD site and a metford-apk site are considered to correspond if they name the same class and the same operator.

Table 5.3 shows, for each subject application, the number of distinct (class, operator) pairs reported by METFORD that are also present in the metford-apk report.

Table 5.3: Class-level mutation site correspondence.

Application	METFORD pairs	Matched	Coverage
AmazeFileManager	221	151	68.3%
AntennaPod	335	269	80.3%
Aegis	226	184	81.4%
KeePassDroid	153	109	71.2%
Omni-Notes	137	99	72.3%
Simplenote	217	125	57.6%

Coverage ranges from 57.6% on Simplenote to 81.4% on Aegis, with a median around 72%. In all subjects, metford-apk reports additional (class, operator) pairs that METFORD does not, which accounts for the higher total mutant counts observed in Section 5.2.

The per-operator breakdown reveals which operators achieve consistent correspondence and which do not. Several operators reach 100% class coverage across all subjects: `FindViewById ReturnsNullMutator`, `InvalidIDFindViewMutator`, `NullIntentMutator`, `NullPut ExtraKeyMutator`, and `NullPutExtraValueMutator`. These operators target specific API call patterns that are unambiguously identifiable at the bytecode level, and the two tools agree on which classes contain them.

`LengthyGUIListenerMutator` consistently shows 0% class coverage: METFORD identifies one to a few classes with this operator in each subject, but metford-apk reports different class names for the same sites. In Java source, anonymous listener classes are declared inside an enclosing class such as `MainActivity`, and METFORD attributes the site to that enclosing class. In smali, the same anonymous class compiles to a separate file with a generated name such as `MainActivity$1`, which is what metford-apk reports. Both tools likely detect the same `onClick` methods, but the class names diverge, causing the class-level comparison to show no overlap.

`BuggyGUIListenerMutator` and `ArithmeticOperatorMutator` show moderate correspondence, typically in the range of 50–85% depending on the subject.

5.4 Time and Space Efficiency

RQ3: Does assembly-level mutation testing improve efficiency and applicability compared to source-level?

Tables 5.4 and 5.5 show the mutation build time and the size of the resulting schemata APK for each subject application where timing data is available.

Table 5.4: Schemata build time.

Application	METFORD	metford-apk
AmazeFileManager	5m29s	11m46s
AntennaPod	7m55s	9m59s
Aegis	1m18s	6m07s
KeePassDroid	4m34s	2m41s
Omni-Notes	4m09s	6m50s
Simplenote	3m18s	5m08s

Build times for metford-apk are generally comparable to metford. On KeePassDroid, metford-apk is faster; on AmazeFileManager it takes roughly twice as long. The difference is largely determined by the number of mutation sites found: more sites means more smali code injected during the weaving phase and a larger input for the subsequent `apktool` repack step. METFORD's build time is dominated by the Gradle compilation of the mutated Java source tree.

Contrary to what might be expected, the metford-apk schemata APK is slightly smaller than the original in every case, by between 2.6% and 7.0%, even though it injects a mutant switch at every mutation site. This is a side effect of the rebuild, not a loss of application code, and it depends on how the input APK was built. Rebuilding a debug APK through `apktool` already makes it smaller even with no mutation applied: a plain rebuild of the AntennaPod APK, injecting nothing, is 7.9% smaller, and the mutated APK is 7.0% smaller, so the difference of about one percentage point is all that the injected mutants add. The effect disappears when the same application is built as a release instead: rebuilding the AntennaPod release APK left it essentially unchanged, at +0.2%. The size change therefore reflects the build type of the input APK and the packaging pipeline, not the mutation.

METFORD's schemata APKs, by contrast, remain within 0.3% of the original, since METFORD mutates the Java source and rebuilds through the standard compiler. Neither tool produces a schemata APK that is impractically large for installation and testing on a device or emulator.

5.5 Mutation Score

RQ3: Does assembly-level mutation testing improve efficiency and applicability compared to source-level?

To measure a mutation score, each mutant in the schemata APK is activated in turn (by setting the `MUTANT_ID` selector through a system property) and the application's own instrumented test suite is run against it. A mutant is counted as *killed* only when it makes a test fail that passes on the original program; tests that already fail on the unmodified application (environment or precondition failures) are therefore ignored. A mutant whose execution does not terminate within a fixed timeout is also counted as killed, since the original program does terminate. Because some test suites are flaky, every killed verdict is re-run and kept only if it reproduces. Mutants that fail

Table 5.5: Schemata APK size.

Application	Orig	METFORD		metford-apk	
	(MB)	MB	%	MB	%
AmazeFileManager	37.3	37.5	+0.3%	35.9	−3.9%
AntennaPod	17.9	17.9	+0.2%	16.6	−7.0%
Aegis	14.2	14.3	+0.3%	13.5	−5.0%
KeePassDroid	11.6	11.6	+0.2%	10.8	−6.7%
Omni-Notes	30.2	30.3	+0.3%	29.4	−2.6%
Simplenote	13.6	13.6	+0.2%	13.1	−3.4%

to load due to invalid generated bytecode are excluded from the denominator, as they are not killed by the test suite.

This evaluation reports two subjects: KeePassDroid, scored over its complete mutant set, and AmazeFileManager, scored over a random sample of 500 of its 1632 mutants. Table 5.6 shows the results alongside the scores reported for the same applications in the original METFORD paper [32], which cover the full mutant set for both subjects.

Table 5.6: Mutation scores for metford-apk and the reference METFORD scores.

Application	Mutants scored	metford-apk	METFORD
KeePassDroid	566 (all)	21.9%	18.2%
AmazeFileManager	500 (sampled)	12.2%	0.1%

For KeePassDroid the two tools are close: metford-apk kills 21.9% of mutants against the 18.2% reported for metford. The small excess is consistent with the operator counts in Section 5.2, since metford-apk generates more mutants overall, including additional arithmetic mutants in the cryptographic code that the unit tests exercise. Several of these are loop-counter mutations that cause non-termination and are caught by the timeout.

For AmazeFileManager the measured score, 12.2%, is far above the 0.1% reported for METFORD. This gap is an artefact of test flakiness rather than a genuine difference in fault detection. Of the 61 mutants counted as killed, 60 fail the exact same group of seven graphical-interface tests, even though those mutants come from twelve different operators acting on unrelated parts of the code. A real detection would be expected to fail different tests for different mutations; an identical failure signature across unrelated mutants indicates that these GUI tests fail in correlated bursts independently of the mutation. The baseline comparison and the reproduce-on-rerun check reduce this effect but do not remove it, because the flaky tests can fail consistently for the duration of a run. This mirrors the flakiness that the original METFORD study also observed on this application.

These two subjects illustrate the two regimes that determine a mutation score in this setting: a stable unit-test suite (KeePassDroid), where metford-apk produces a score close to the source-level tool, and a flaky graphical-interface suite (AmazeFileManager), where the score is dominated

by non-deterministic test behaviour. A broader evaluation across more subjects, with stronger flakiness control, is left for future work.

5.6 Applicability to Closed-Source Applications

RQ3 (applicability): Can metford-apk mutate applications for which no source code is available?

Because metford-apk operates directly on a compiled APK, it can mutate applications whose source code is not available. This is not possible for METFORD, which requires the Java source tree. The three closed-source subjects therefore cannot be compared against METFORD as in Sections 5.2–5.4, and, lacking published instrumented test suites, cannot be assigned a mutation score as in Section 5.5. What can be demonstrated is that the full generation pipeline — decompilation, mutation-site identification, schemata weaving, and repackaging — runs end-to-end on an arbitrary released APK, and how many mutants it produces.

Table 5.7 reports the results. The per-operator breakdown is given in Appendix A.

Table 5.7: Generation results for the closed-source subjects (metford-apk only).

Application	Original APK	Mutants	Schemata APK
CleanCalculator	0.7 MB	31	0.7 MB
CPU-Z	5.1 MB	46	5.3 MB
Musicolet	32.1 MB	2184	32.2 MB

CleanCalculator and CPU-Z yield few mutants. CleanCalculator is a minimal calculator with little application code. CPU-Z, although 5 MB, exposes only 46 mutation sites because its functionality is implemented largely in native C/C++ libraries reached through JNI; the Java bytecode layer that metford-apk can mutate is a thin wrapper around them.

Musicolet, by contrast, yields 2184 mutants — comparable to the largest open-source subject, AntennaPod (2456), and far above the other two closed-source applications. Its user interface and playback logic are implemented in native Android Java with custom `View` subclasses, which the view- and listener-oriented operators target directly: the `FindViewById`, `InvalidIDFindView`, `InvalidViewFocus`, and `ViewComponentNotVisible` operators alone account for over a thousand mutants, and `ArithmeticOperator` for a further 471. This confirms that the approach scales to a substantial, real-world closed-source application rather than only to trivial ones.

The three subjects together illustrate a general property: mutant yield is governed by the amount of mutable Java bytecode, not by APK size. CPU-Z (5 MB) yields 46 mutants while Musicolet (32 MB) yields 2184, because Musicolet's logic lives in Java whereas CPU-Z's — and most of its size — lies in native libraries and assets. Applications built as thin wrappers over WebViews or native engines therefore expose few mutation sites regardless of their download size, and are poor candidates for bytecode-level mutation.

The schemata APK remains close to the original size in all three cases: the injected switch code adds under one megabyte of bytecode. For Musicolet this is negligible against a 32 MB APK dominated by native audio libraries, giving a bloat factor of 1.01, compared with the 1.5–1.9 observed for the smaller open-source subjects whose size is dominated by bytecode (Section 5.4). One methodological note applies: for CleanCalculator and CPU-Z the schemata APK was produced by the standard metford-apk repackaging step, whereas for Musicolet the version of `apktool` used fails to re-link two library resource attributes unrelated to the mutation, so its schemata APK was assembled by replacing the mutated `classes.dex` in the original archive. Both routes package the same mutated bytecode; only the resource-repackaging path differs.

5.7 Limitations

5.7.1 Subject application size

All the subject applications are small or medium-sized, and larger applications could not be used. This is a limitation of metford-apk rather than a deliberate choice of subjects.

The cause lies in the schemata strategy. metford-apk embeds every mutant into a single APK, so that all of them can be evaluated from one instrumented build. For a large application the number of mutation sites is high, and packing all the corresponding mutants into one APK runs into the structural limits of the DEX format: an individual Dalvik method has a maximum bytecode size, and a DEX file has a fixed ceiling on the number of methods and string references it can hold. When these limits are reached, the schemata APK can no longer be built, and the application cannot be processed at all.

This constraint is specific to metford-apk's post-compilation model. A source-level schemata tool such as METFORD injects its mutants into the Java source and then hands the result to the normal Android build pipeline, which applies multidex automatically: when the method, field, or string counts exceed a single DEX's ceiling, the compiler splits the output across several DEX files (`classes.dex`, `classes2.dex`, and so on) without any intervention. metford-apk, by contrast, operates on an already-compiled APK disassembling to smali, injecting mutant branches, and repacking after the point at which the build would have performed this splitting. Because it does not re-run that partitioning, the accumulated mutant code must fit within the DEX limits as they already stand, and once a method's bytecode size or a DEX's method or string count is exceeded, the repack fails.

As a result, the evaluation is restricted to applications small enough for the full mutant set to fit within these limits. This affects how far the results generalise, since larger and more complex codebases were not exercised. A possible way to lift this restriction is to split the mutants across several APKs, each covering a disjoint subset of operators or classes; this is discussed as future work in Section 6.2.

5.7.2 Coverage metric granularity

The coverage metric used in RQ2 measures the correspondence between metford-apk and METFORD at the class level, which is only a coarse proxy for true site-level correspondence. Two tools may mutate the same class at different locations and still be counted as matching. A finer comparison would require matching mutation sites by source line, but this is not always possible because Dalvik compilers emit one `.line` directive per Java source line rather than per smali instruction, so a mutation anchor that falls on a later instruction from the same source line has no preceding directive and its line number is null. This is a general property of Dalvik bytecode, not specific to the applications used in this study. The coverage numbers reported in this work should therefore be read as an approximation rather than an exact measure of overlap. A more precise metric is discussed as future work in Section 6.2.

5.7.3 Skipped mutation sites

metford-apk skips some mutation sites. The tool adds a local register to apply a mutation, and when this would push a parameter register beyond `v15` the site is skipped to keep the method valid. These skipped sites lower the coverage metford-apk can achieve relative to metford, so part of the gap reported in RQ2 is due to this constraint rather than to a real difference in fault model. A way to recover these sites through improved register allocation is discussed as future work in Section 6.2

5.7.4 WebView and non-Dalvik code

metford-apk operates on Dalvik bytecode and is therefore limited to the native layer of Android applications. Applications that implement part or all of their logic inside a WebView, executing JavaScript rather than Dalvik instructions, are not covered: the tool has no visibility into the web assets bundled in the APK and cannot inject mutations into them. Similarly, applications that use native libraries compiled to ARM or x86 machine code are not mutated at that layer. For such applications, mutation testing of the non-Dalvik components would require a separate tool targeting the appropriate representation.

5.7.5 Mutation score reliability

The mutation scores in RQ3 are limited in two ways. First, only two applications are scored, and one of them (AmazeFileManager) is scored over a random sample of its mutants rather than the full set, to keep the cost of running the test suite against each mutant manageable; a sampled score is an estimate that may vary with the chosen sample. Second, the scores are affected by flaky tests, which can fail even when the mutation is not the cause and make the score look higher than it really is, as discussed in Section 5.5. Evaluating more applications and handling flaky tests better would improve these results.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This work presented metford-apk, a schemata mutation testing tool for Android that operates directly on Dalvik bytecode. Rather than requiring Java or Kotlin source code, the tool works from a compiled APK, making it applicable to any Android application regardless of whether its source is available. Fourteen mutation operators drawn from the METFORD operator set were implemented at the assembly level, covering traditional arithmetic faults and Android-specific fault patterns involving Intent communication and graphical user interface behaviour.

The evaluation compared metford-apk against METFORD across six open-source Android applications. metford-apk consistently generated more mutants than METFORD, with totals ranging from 7% more on Aegis to 90% more on AntennaPod. At the class level, metford-apk covered between 57% and 81% of the (class, operator) pairs identified by METFORD, with a median around 72%. Several operators achieved complete class-level correspondence across all subjects, including `NullIntentMutator`, `FindViewByIdReturnsNullMutator`, `InvalidIDFindViewMutator`, `NullPutExtraKeyMutator`, and `NullPutExtraValueMutator`. Build times were broadly comparable between the two tools, and the size of the schemata APK scaled with the number of mutants injected.

The additional mutants generated by metford-apk are concentrated in operators that METFORD scopes narrowly at the source level, such as `FindViewByIdReturnsNullMutator` and `LengthyGUIListenerMutator`. At the bytecode level, these operators are applied to all matching call sites regardless of the enclosing class hierarchy, which produces broader but potentially noisier coverage.

The tool was also applied to three closed-source applications, demonstrating that assembly-level mutation testing can reach targets that are entirely inaccessible to source-level tools. This confirms the original motivation for the work: a bytecode-level tool is applicable in a strictly wider range of contexts than a source-level tool.

Regarding the research questions posed in the introduction: the schemata strategy is fully applicable at the assembly level using Dalvik's `packed-switch` instruction, with the constraint

that fall-through semantics replace the explicit default jump of the JVM. Fourteen source-level operators were found to remain meaningful at the assembly level, requiring only minor adaptations for missing opcodes and register constraints. Assembly-level mutation testing improves applicability by removing the source code requirement; build times are competitive with the source-level approach, and the resulting schemata APK remains installable on standard devices.

6.2 Future Work

Several directions remain open for future investigation.

Native application scope. metford-apk operates on Dalvik bytecode and is therefore limited to the native layer of Android applications. Applications that implement their logic inside a WebView, executing JavaScript rather than Dalvik instructions, are not covered by the tool. For such applications, a JavaScript-level mutation testing tool applied to the bundled web assets would be the appropriate complement.

Equivalent mutant detection. Some of the mutants generated by metford-apk may be equivalent to the original program, meaning no test can distinguish them. Detecting or estimating the proportion of equivalent mutants is important for interpreting mutation scores accurately.

Operator correspondence refinement. The class-level coverage metric used in this work is a coarse proxy for true site correspondence. A finer comparison could be achieved by improving line number attribution in the mutation report. The current limitation arises from missing `.line` directives in smali; a possible remedy is to map smali bytecode offsets back to source lines using the debug information encoded in the DEX file.

Improved register allocation. The register safety check currently skips mutation sites in methods where adding a local register would push a parameter register above `v15`. An alternative is to improve register allocation, for example by performing liveness analysis and compacting live ranges so that the extra local can reuse a slot that is not live at the mutation site. This would recover coverage at skipped sites without changing the method's semantics.

Additional operators. The current operator set mirrors METFORD's fourteen operators. Additional operators relevant to Android, such as mutations targeting lifecycle callbacks, permission checks, or asynchronous task handling, could be added to improve fault model coverage.

Schemata size limits. Embedding all mutants in a single APK has practical limits. Dalvik methods have a maximum bytecode size, and DEX files have limits on the number of methods and string references. For large applications with many mutation sites, the schemata APK may approach

these limits. A possible mitigation is to partition the mutant set across multiple APKs, each covering a disjoint subset of operators or classes, at the cost of requiring multiple instrumented builds per evaluation run.

Broader subject set. The evaluation used six open-source and three closed-source applications. Extending the evaluation to a larger and more diverse set of subjects, including applications with larger codebases or more complex build configurations, would strengthen the robustness of the results.

References

[1] João Bispo and João MP Cardoso. "Clava: C/C++ source-to-source compilation using LARA". In: *SoftwareX* 12, 100565 (2020). DOI: 10.1016/j.softx.2020.100565. URL: https://www.sciencedirect.com/science/article/pii/S2352711019302122.

[2] Tiago Carvalho et al. "A DSL-based runtime adaptivity framework for Java". In: *SoftwareX* 23, 101496 (2023). DOI: 10.1016/j.softx.2023.101496. URL: https://www.sciencedirect.com/science/article/pii/S2352711023001929.

[3] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. "Automated Test Input Generation for Android: Are We There Yet? (E)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). Nov. 2015, pp. 429–440. DOI: 10.1109/ASE.2015.89. URL: https://ieeexplore.ieee.org/document/7372031 (visited on 11/03/2025).

[4] Lin Deng. "Mutation Testing for Android Applications". Ph.D. Dissertation. Fairfax, VA: George Mason University, 2017.

[5] Lin Deng and Jeff Offutt. "Experimental Evaluation of Redundancy in Android Mutation Testing". In: *International Journal of Software Engineering and Knowledge Engineering* 28.11 (Nov. 2018). Publisher: World Scientific Pub Co Pte Lt, pp. 1597–1618. ISSN: 0218-1940. DOI: 10.1142/s0218194018400193. URL: http://dx.doi.org/10.1142/S0218194018400193.

[6] Lin Deng and Jeff Offutt. "Reducing the Cost of Android Mutation Testing". In: *International Conferences on Software Engineering and Knowledge Engineering*. Vol. 2018. ISSN: 2325-9000. KSI Research Inc. and Knowledge Systems Institute Graduate School, July 1, 2018, pp. 542–591. DOI: 10.18293/seke2018-184. URL: http://dx.doi.org/10.18293/SEKE2018-184.

[7] Lin Deng, Jeff Offutt, and David Samudio. "Is Mutation Analysis Effective at Testing Android Apps?" In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). July 2017, pp. 86–93. DOI: 10.1109/QRS.2017.19. URL: https://ieeexplore.ieee.org/document/8009912/ (visited on 10/29/2025).

[8] Lin Deng et al. "Mutation operators for testing Android apps". In: *Information and Software Technology* 81 (Jan. 1, 2017), pp. 154–168. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.04.012. URL: https://www.sciencedirect.com/science/article/pii/S0950584916300684 (visited on 10/29/2025).

[9] Lin Deng et al. “Towards mutation analysis of Android apps”. In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). Apr. 2015, pp. 1–10. DOI: 10.1109/ICSTW.2015.7107450. URL: https://ieeexplore.ieee.org/abstract/document/7107450 (visited on 10/30/2025).

[10] Camilo Escobar-Velasquez. “Source-Codeless Testing for Android Apps”. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). ISSN: 2159-4848. Oct. 2020, pp. 433–435. DOI: 10.1109/ICST46399.2020.00057. URL: https://ieeexplore.ieee.org/abstract/document/9159057 (visited on 11/11/2025).

[11] Camilo Escobar-Velasquez, Michael Osorio-Riano, and Mario Linares-Vasquez. “MutAPK: Source-Codeless Mutant Generation for Android Apps”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Nov. 2019. DOI: 10.1109/ase.2019.00109. URL: http://dx.doi.org/10.1109/ASE.2019.00109.

[12] Camilo Escobar-Velasquez et al. “Enabling Mutant Generation for Open- and Closed-Source Android Apps”. In: *IEEE Transactions on Software Engineering* 48.1 (Jan. 1, 2022). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 186–208. ISSN: 0098-5589. DOI: 10.1109/tse.2020.2982638. URL: http://dx.doi.org/10.1109/tse.2020.2982638.

[13] Camilo Escobar-Velásquez, Diego Riveros, and Mario Linares-Vásquez. “MutAPK 2.0: a tool for reducing mutation testing effort of Android apps”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, Nov. 8, 2020, pp. 1611–1615. ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3417942. URL: https://doi.org/10.1145/3368089.3417942 (visited on 10/29/2025).

[14] Reyhaneh Jabbarvand and Sam Malek. “ţDroid: an energy-aware mutation testing framework for Android”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, Aug. 21, 2017, pp. 208–219. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3106244. URL: https://dl.acm.org/doi/10.1145/3106237.3106244 (visited on 11/03/2025).

[15] Yue Jia and Mark Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *IEEE Transactions on Software Engineering* 37.5 (Sept. 2011), pp. 649–678. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.62. URL: http://ieeexplore.ieee.org/document/5487526/ (visited on 10/01/2025).

[16] Pingfan Kong et al. “Automated Testing of Android Apps: A Systematic Literature Review”. In: *IEEE Transactions on Reliability* 68.1 (Mar. 2019), pp. 45–66. ISSN: 1558-1721. DOI: 10.1109/TR.2018.2865733.

[17] Pedro Henrique Kuroishi et al. “Designing Mutation Operators for Android Device Components: A View Through Bluetooth and Location APIs”. In: *Anais do XXXIX Simp\' osio Brasileiro de Engenharia de Software (SBES 2025)*. Sociedade Brasileira de Computaç~

ao, Sept. 22, 2025, pp. 149–159. DOI: 10.5753/sbes.2025.9878. URL: http://dx.doi.org/10.5753/sbes.2025.9878.

[18] Jian Liu et al. "DroidMutator". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. ACM, June 27, 2020, pp. 77–80. DOI: 10.1145/3377812.3382134. URL: http://dx.doi.org/10.1145/3377812.3382134.

[19] Eduardo Luna and Omar El Ariss. "Edroid: A Mutation Tool for Android Apps". In: *2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT)*. 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT). Oct. 2018, pp. 99–108. DOI: 10.1109/CONISOFT.2018.8645883. URL: https://ieeexplore.ieee.org/abstract/document/8645883 (visited on 10/30/2025).

[20] Kevin Moran et al. "MDroid+: a mutation testing framework for android". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. New York, NY, USA: Association for Computing Machinery, May 27, 2018, pp. 33–36. ISBN: 978-1-4503-5663-3. DOI: 10.1145/3183440.3183492. URL: https://doi.org/10.1145/3183440.3183492 (visited on 10/29/2025).

[21] Ana C. R. Paiva et al. "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)*. 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). Apr. 2019, pp. 102–111. DOI: 10.1109/ICSTW.2019.00038. URL: https://ieeexplore.ieee.org/document/8728917 (visited on 11/03/2025).

[22] Mike Papadakis et al. "Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 1. ISSN: 1558-1225. May 2015, pp. 936–946. DOI: 10.1109/ICSE.2015.103. URL: https://ieeexplore.ieee.org/document/7194639 (visited on 11/03/2025).

[23] Pedro Pinto et al. "Aspect composition for multiple target languages using LARA". In: *Computer Languages, Systems & Structures* 53 (2018), pp. 1–26.

[24] Ibrahim Anka Salihu et al. "Mutation Testing Techniques for Android Applications: A Comparative Study". In: *2023 2nd International Conference on Multidisciplinary Engineering and Applied Science (ICMEAS)*. 2023 2nd International Conference on Multidisciplinary Engineering and Applied Science (ICMEAS). Vol. 1. Nov. 2023, pp. 1–5. DOI: 10.1109/ICMEAS58693.2023.10429866. URL: https://ieeexplore.ieee.org/abstract/document/10429866 (visited on 10/30/2025).

[25] Gustavo Amorim Santos. "Detecting Resource Leaks on Android with Alpakka". In: (July 12, 2024). Accepted: 2025-02-03T02:44:45Z. URL: https://hdl.handle.net/10216/161050 (visited on 11/05/2025).

[26] Henrique Neves Silva et al. "A mapping study on mutation testing for mobile applications". In: *Software Testing, Verification and Reliability* 32.8 (2022), e1801. ISSN: 1099-1689. DOI: 10.1002/stvr.1801. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1801 (visited on 10/30/2025).

[27] Statista. *Mobile app downloads worldwide 20182024*. en. Accessed: 2025-10-10. Statista. 2024. URL: https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/.

[28] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. "Mutation Analysis Using Mutant Schemata". In: *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '93. New York, NY, USA: ACM, 1993, pp. 139–148. DOI: 10.1145/154183.154265.

[29] Macario Polo Usaola et al. "An Architecture for the Development of Mutation Operators". In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Mar. 2017, pp. 143–148. DOI: 10.1109/ICSTW.2017.31.

[30] Escobar Velásquez and Camilo Andrés. "Automatic analysis of Android closed-source apps to support software engineering tasks : the mutation testing case". In: (2019). Publisher: Uniandes. URL: https://hdl.handle.net/1992/34998 (visited on 10/30/2025).

[31] Escobar Velásquez and Camilo Andrés. "On improving analysis and testing of open- and closed-source Android Apps". In: (June 30, 2023). Publisher: Universidad de los Andes. DOI: 10.57784/1992/68809. URL: https://hdl.handle.net/1992/68809 (visited on 11/11/2025).

[32] Auri M.R. Vincenzi et al. "METFORD Mutation tEsTing Framework fOR anDroid". In: *Journal of Systems and Software* 222 (Apr. 2025). Publisher: Elsevier BV, p. 112332. ISSN: 0164-1212. DOI: 10.1016/j.jss.2024.112332. URL: http://dx.doi.org/10.1016/j.jss.2024.112332.

[33] Yuan Wei. "MuDroid : Mutation Testing for Android Apps Undergraduate Final Year Individual Project". In: (2016).

Appendix A

Per-Operator Mutant Counts

This appendix lists the per-operator mutant counts for each subject application, comparing METFORD and metford-apk. The trends are discussed in Section 5.2. Operator names are shown without the common `Mutator` suffix.

Table A.1: Per-operator mutant counts for Aegis.

Operator	METFORD	metford-apk	diff
ArithmeticOperator	220	180	-40
BuggyGUIListener	97	46	-51
FindViewByIdReturnsNull	59	134	+75
IntentTargetReplacement	0	19	+19
InvalidIDFindView	68	134	+66
InvalidKeyIntent	30	19	-11
InvalidViewFocus	139	135	-4
LengthyGUICreation	17	15	-2
LengthyGUIListener	2	50	+48
NullIntent	28	29	+1
NullPutExtraKey	36	38	+2
NullPutExtraValue	36	38	+2
RandomIntentAction	108	58	-50
ViewComponentNotVisible	139	135	-4
TOTAL	979	1030	+51

The closed-source subjects are not compared against METFORD (no source is available); their per-operator counts for metford-apk alone are given in Table A.7. They are discussed in Section 5.6.

Table A.2: Per-operator mutant counts for AmazeFileManager.

Operator	METFORD	metford-apk	diff
ArithmeticOperator	326	620	+294
BuggyGUIListener	45	52	+7
FindViewByIdReturnsNull	28	135	+107
IntentTargetReplacement	0	29	+29
InvalidIDFindView	34	135	+101
InvalidKeyIntent	41	32	-9
InvalidViewFocus	145	145	0
LengthyGUICreation	18	16	-2
LengthyGUIListener	0	70	+70
NullIntent	40	49	+9
NullPutExtraKey	37	53	+16
NullPutExtraValue	37	53	+16
RandomIntentAction	144	98	-46
ViewComponentNotVisible	145	145	0
TOTAL	1040	1632	+592

Table A.3: Per-operator mutant counts for AntennaPod.

Operator	METFORD	metford-apk	diff
ArithmeticOperator	321	663	+342
BuggyGUIListener	186	128	-58
FindViewByIdReturnsNull	15	312	+297
IntentTargetReplacement	0	24	+24
InvalidIDFindView	33	312	+279
InvalidKeyIntent	19	24	+5
InvalidViewFocus	258	325	+67
LengthyGUICreation	20	20	0
LengthyGUIListener	1	141	+140
NullIntent	15	32	+17
NullPutExtraKey	14	43	+29
NullPutExtraValue	14	43	+29
RandomIntentAction	62	64	+2
ViewComponentNotVisible	258	325	+67
TOTAL	1216	2456	+1240

Table A.4: Per-operator mutant counts for KeePassDroid.

Operator	METFORD	metford-apk	diff
ArithmeticOperator	132	205	+73
BuggyGUIListener	14	11	-3
FindViewByIdReturnsNull	5	60	+55
IntentTargetReplacement	0	7	+7
InvalidIDFindView	5	60	+55
InvalidKeyIntent	13	7	-6
InvalidViewFocus	75	70	-5
LengthyGUICreation	16	13	-3
LengthyGUIListener	17	10	-7
NullIntent	10	15	+5
NullPutExtraKey	6	4	-2
NullPutExtraValue	6	4	-2
RandomIntentAction	32	30	-2
ViewComponentNotVisible	75	70	-5
TOTAL	406	566	+160

Table A.5: Per-operator mutant counts for Omni-Notes.

Operator	METFORD	metford-apk	diff
ArithmeticOperator	71	88	+17
BuggyGUIListener	18	19	+1
FindViewByIdReturnsNull	11	66	+55
IntentTargetReplacement	0	15	+15
InvalidIDFindView	16	66	+50
InvalidKeyIntent	15	16	+1
InvalidViewFocus	33	66	+33
LengthyGUICreation	12	11	-1
LengthyGUIListener	4	21	+17
NullIntent	18	17	-1
NullPutExtraKey	14	8	-6
NullPutExtraValue	14	8	-6
RandomIntentAction	62	34	-28
ViewComponentNotVisible	33	66	+33
TOTAL	321	501	+180

Table A.6: Per-operator mutant counts for Simplenote.

Operator	METFORD	metford-apk	diff
ArithmeticOperator	114	136	+22
BuggyGUIListener	50	40	-10
FindViewByIdReturnsNull	13	264	+251
IntentTargetReplacement	0	35	+35
InvalidIDFindView	15	264	+249
InvalidKeyIntent	44	37	-7
InvalidViewFocus	104	270	+166
LengthyGUICreation	12	12	0
LengthyGUIListener	45	56	+11
NullIntent	42	14	-28
NullPutExtraKey	53	24	-29
NullPutExtraValue	53	24	-29
RandomIntentAction	155	28	-127
ViewComponentNotVisible	104	270	+166
TOTAL	804	1474	+670

Table A.7: Per-operator mutant counts for the closed-source subjects (metford-apk only).

Operator	CleanCalculator	CPU-Z	Musicolet
ArithmeticOperator	8	4	471
BuggyGUIListener	0	6	69
FindViewByIdReturnsNull	0	8	269
IntentTargetReplacement	2	0	0
InvalidIDFindView	0	8	225
InvalidKeyIntent	2	1	138
InvalidViewFocus	0	8	283
LengthyGUICreation	2	2	25
LengthyGUIListener	0	1	18
NullIntent	5	0	79
NullPutExtraKey	1	0	83
NullPutExtraValue	1	0	83
RandomIntentAction	10	0	158
ViewComponentNotVisible	0	8	283
TOTAL	31	46	2184