WallMauer: Robust Code Coverage Instrumentation for Android Apps

Michael Auer
University of Passau
Passau, Germany

Iván Arcuschin Moreno
University of Buenos Aires
Buenos Aires, Argentinia

Gordon Fraser
University of Passau
Passau, Germany

ABSTRACT

Code coverage is the primary metric used to assess the quality of test suites, and it is the foundation of many automated techniques ranging from fault localization to search-based optimization approaches. Code coverage is measured by inserting probes into programs which keep track of executed code when running tests. While this can be easily done in many testing domains, it remains a challenging task for Android apps, mainly due to the nature of the Dalvik bytecode used for Android apps: First, the internal handling of registers inhibits common types of probes. To circumvent this problem, existing tools often rely on conversion of Dalvik bytecode to standard Java bytecode or source code, but during the conversion back to Dalvik bytecode errors and inconsistencies may occur. Furthermore, a strict limit of the number of methods and classes contained in a single archive of Dalvik bytecode (*DEX* file) requires spliting apps into multiple such DEX files (*multidex* approach), which is rarely supported by existing coverage instrumentation frameworks. This is not only a problem when trying to instrument regular *multidex* apps, but the coverage instrumentation itself increases the number of methods, potentially requiring a *multidex* solution even for apps that would otherwise fit in a single DEX file. In this paper we present WallMauer, a new code coverage tool that overcomes these limitations: It supports *multidex*, and avoids inconsistencies by rigorously instrumenting Dalvik bytecode directly. WallMauer solely requires an APK file as input and as such it can be easily integrated into any existing testing environment. Using a set of 1000 open source apps from the F-Droid repository we demonstrate that WallMauer is extremely robust, successfully instrumenting more than 99% of apps, more than any other state-of-the-art instrumentation framework.

KEYWORDS

Instrumentation, Code Coverage, Android

ACM Reference Format:
Michael Auer, Iván Arcuschin Moreno, and Gordon Fraser. 2024. WallMauer: Robust Code Coverage Instrumentation for Android Apps. In *5th ACM/IEEE International Conference on Automation of Software Test (AST 2024) (AST '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3644032.3644462

AST '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0588-5/24/04
https://doi.org/10.1145/3644032.3644462

1 INTRODUCTION

Code coverage is one of the central metrics in software testing. It serves not only to assess the quality of tests, but also to inform analysis techniques such as fault localization, or to guide search-based algorithms towards generating coverage-optimized test suites (e.g., [9, 23, 29]). Measuring code coverage consists of inserting measurement probes into a program at the level of granularity desired (e.g., statements or methods), and then running the tests on the instrumented program. The resulting coverage information collected by the probes can then be visualized for users or processed by automated techniques. While this process can easily and reliably be implemented in many domains, it is challenging to apply on Android apps, in particular when the source code is not available.

Adding coverage instrumentation to Android apps without source code is a common scenario for many applications, in particular in research. Consequently, a number of such black-box code coverage tools have been proposed for Android. Some of these consider only a coarse level of granularity such as method coverage, since the instrumentation required for this is simpler. Those that target finer granularity face challenges that ultimately mean they will only work on a subset of apps successfully. There are multiple reasons for this, rooted in the Dalvik bytecode used for Android apps:

- Dalvik bytecode imposes harsh limitations on how instructions can leverage registers, which makes modifications to the bytecode complicated. Instrumentation tools that fail to adhere to these rules may produce malformed bytecode.
- To avoid this problem, some tools perform their instrumentation at a different level of abstraction, e.g., at the Java bytecode level or source code level. However, the conversion from Dalvik bytecode to another abstraction and the corresponding re-conversion later on is in most cases not lossless and may again result in bytecode that is corrupted or diverges from the original code.
- The biggest challenge is the 65K method limit imposed on DEX files, i.e., binaries containing the executable Dalvik bytecode in the form of classes and methods. If this limit is exceeded, the bytecode needs to be split into multiple DEX files. This support was introduced with API level 21 and is known as *multidex* [5]. Even if existing Android apps do not use a *multidex* approach already, the instrumentation itself is likely to introduce additional classes and methods potentially requiring a split of the modified DEX file. However most state-of-the-art instrumentation tools do not support *multidex*, which limits their applicability.

To overcome these issues, in this paper we introduce WallMauer, a tool that overcomes these challenges by (1) directly operating at the Dalvik bytecode level rather than requiring a conversion to another abstraction level, (2) following the Dalvik bytecode

specification very strictly, thus producing only valid bytecode, and (3) supporting *multidex* out of the box, thus being applicable to any apps. WallMauer comes as a Java-based command-line application and requires nothing more than the APK file of an app to instrument, i.e., the standard Android package format. WallMauer can be easily integrated into any existing testing environment and has already been successfully applied in combination with the open-source test generator Mate [14] by instrumenting hundreds of Android applications in multiple empirical studies, e.g., [11, 12].

In detail, this paper makes the following contributions:

- We provide a robust code coverage instrumentation tool called WallMauer for Android apps that requires only the APK file as input and can be easily integrated into any existing testing environment.
- We empirically demonstrate the robustness of WallMauer on a set of 1000 F-Droid applications and compare it to state-of-the-art tools ACVTool and COSMO.
- We empirically evaluate the overhead induced by the instrumentation and show it is able to compete with ACVTool and COSMO.
- We demonstrate that the reported coverage is compliant with the results obtained from ACVTool and transitively also to COSMO.
- We make WallMauer publicly available on GitHub (https://github.com/mate-android-testing/wallmauer) to support the Android testing and research community.

The results of our empirical study show that WallMauer outperforms state-of-the-art tools ACVTool and COSMO in terms of successfully instrumented apps, while the overhead induced by the instrumentation is largely negligible and magnitudes lower than both other tools. In detail, WallMauer can produce an instrumented APK for 999 out of 1000 F-Droid applications from which 998 remain healthy after being automatically tested, while ACVTool manages to instrument 885 apps from which 638 could be successfully explored, while COSMO can handle 891 apps of which 793 appear to be functional after being tested.

2 BACKGROUND

2.1 Code Coverage Analysis

The most common way to determine code coverage is by instrumenting the code of the program under test by inserting probes at specific locations, depending on the targeted level of granularity, e.g., at method, basic block or statement level [19]. The tests are then executed on the resulting instrumented version of the program, while the probes collect information about which parts of the code were covered and typically store this information in trace files. From these, a code coverage report can be generated and visualized for users, and the information can also directly inform other tools and approaches, such as automated test generation tools.

The instrumentation can be done at different levels of abstraction, e.g., on source code or bytecode level. It is important to note that the reported code coverage measured at different abstraction levels may not be interchangeable [16, 26], e.g., a single source code statement typically corresponds to multiple bytecode instructions. Besides technical questions, the choice of abstraction level is also influenced by the targeted application of the code coverage instrumentation: While it is reasonable to assume that a developer running unit tests on their own code has the source code at hand, such that it can be directly instrumented at source level, there are many scenarios where availability of source code cannot be assumed. For example, many testing or fuzzing scenarios are applied to Android apps packaged as APK files, with only Dalvik bytecode available. This is also the scenario we consider in this paper. Unfortunately, instrumenting at bytecode level on Android is non-trivial.

2.2 Android Components

Android apps are composed of four main types of components, Activities, Services, Broadcast Receivers and Content Providers [1]: Activities are the main entry point for the user. They are responsible for displaying the user interface and handling user interactions (e.g., login screen). Services are background components that perform long-running operations without a user interface (e.g., file download). Broadcast Receivers are components that receive and react to system-wide broadcast announcements (e.g., low battery). Content Providers are components that manage access to a structured set of data. They encapsulate the data and provide mechanisms for defining data security (e.g., contacts). Since all of these types of components consist of source code that may affect a program's behaviour and may be targeted by tests, coverage instrumentation needs to be applicable to any of them.

2.3 Compilation Process

Android development officially supports Java and Kotlin as programming languages. The Android SDK provides a set of libraries and tools that are required to build, test and debug Android apps. The compilation process consists of two steps: First, the Java or Kotlin source code is compiled to Java bytecode. Second, the Java bytecode is translated to Dalvik bytecode. The Android platform then runs this Dalvik bytecode in the Android Runtime (ART) environment. ART is a register-based virtual machine that executes Dalvik bytecode. Consequently, collecting coverage information requires probes within the executed Dalvik bytecode, but these probes may be inserted at any abstraction level from source code to Dalvik bytecode directly. If source code is not available, Dalvik bytecode can be converted back to Java bytecode, although a round-trip conversion back to Dalvik may not be lossless.

2.4 APK Files

Android applications are distributed as APK (Android Package) files. An APK file is a ZIP archive that contains the Dalvik bytecode, the Android manifest and all resources of the app [8]. The Android manifest is an XML file that contains information about the app and its components. It also declares the list of permissions that the app requires to run. Permissions are used to protect sensitive data and functionality. For example, the `READ_CONTACTS` permission is required to read the user's contacts.

2.5 Dalvik Bytecode

During the compilation process, Java bytecode is compiled into Dalvik bytecode. Although converting back from Dalvik to Java bytecode is possible, this transformation is not lossless and the resulting Java bytecode may not be equivalent to the original one,

```
.method private interpretBMI(D)Ljava/lang/String;
.registers 6

const-wide v0, 0x4032800000000000L
cmpg-double v2, p1, v0

if-gez v2, :cond_1
const-string p1, "You are underweight"
return-object p1

:cond_1
const-wide/high16 v0, 0x4039000000000000L
cmpg-double v2, p1, v0

if-gez v2, :cond_2
const-string p1, "You are normal weight"
return-object p1

:cond_2
const-string p1, "You are overweight"
return-object p1

.end method
```

Listing 1: Smali representation of an arbitrary method.

e.g., switch tables are encoded differently and cannot be mapped 1:1. Dalvik bytecode can also be converted to Smali, a human-readable representation of Dalvik bytecode used by reverse engineers to analyze Android apps.

Inside APKs, Dalvik bytecode is stored in DEX (Dalvik Executable) files. A DEX file contains a set of classes and their associated methods. One of the limitations of this format is that each DEX file can only contain up to 65,536 methods (known as the 65K method limit) [5]. To overcome this limitation, the Android platform allows developers to split the Dalvik bytecode into multiple DEX files inside an APK. This support is known as *multidex*.

Unlike the Java Virtual Machine (JVM), which is stack-based, the ART environment is register-based [2], i.e., the values of the variables used by the program are stored in a set of registers instead of a stack. The ART environment has 32-bit wide registers to store primitive types like integers or floating-point numbers. For 64-bit data types like double or long two adjacent registers are required. The maximal number of registers per method is limited to a number of 65,536. However, due to optimizations most instructions can only reference the first 256 registers or even the first 16 registers, e.g., the regular *invoke* instruction can only address the first 16 registers. While many instructions disregard the register type, there are certain instructions that assume a specific type, e.g., the *move-object* instruction is dedicated to shift object types while the regular *move* instruction can shift any primitive type.

There are certain instructions that always come paired, e.g., any *invoke* instruction that is expected to return some result other than *void* is directly followed by a *move-result* instruction. It is not allowed to place any instructions in between.

Each method needs to declare the number of required registers where a distinction is made between local and parameter registers. The local registers come first and are denoted *v0, v1, ...* whereas the parameter registers follow and are denoted as *p0, p1, ...* Consider the Smali representation of an arbitrary method in Listing 1: The method declares six registers where three are local and three are parameter registers. It takes an explicit double parameter (*D*) and since the method is not static, it also has an implicit parameter that contains the *this*-reference, i.e., *p0* refers to the *this*-reference and *p1* and *p2* contain the upper and low half of the double value, respectively. Note that *p2* is never explicitly referenced in the code since any instruction involving a wide type like double or long only address the upper half, i.e., *p1* in the example. Likewise, the local register *v1* containing the lower half of a long value is never explicitly used but instead only *v0*. Although the local and parameter registers are differently denoted they are internally labelled from 0 to N. That means the parameter register *p0* could also be regarded as *v4*, *p1* as *v5* and so on.

Any modifications to the bytecode need to ensure that the individual instructions still reference only eligible registers. For example, the introduction of an additional local register would increase the internal ids of the parameter registers by one. If any of the parameter registers now has an id greater than 16, corresponding *invoke* bytecode instructions would not be valid anymore. In this case the parameter register needs to be first shifted to a register with a lower id before being used by the *invoke* instruction. This makes the instrumentation at the Dalvik bytecode a complex endeavor.

2.6 Android Coverage Instrumentation

Several tools have been proposed over the years for measuring code coverage of Android apps. These tools can be classified into two categories: White-box instrumentation and black-box instrumentation [22]. White-box instrumentation tools, e.g., Emma [4], JaCoCo [7] or the IntelliJ IDEA code coverage runner [6] assume the presence of the source code and modify either the source code directly or the intermediate Java bytecode. In contrast, black-box instrumentation tools like InsDal [20], CovDroid [28] or ACVTool [22] assume only the APK file as input and directly operate on the Dalvik bytecode or any intermediate representation, e.g., Smali. In this paper we focus on black-box approaches which do not make the assumption of having source code available. Consequently, these approaches need to be able to insert coverage probes into Dalvik bytecode.

There are two main approaches for instrumenting Dalvik bytecode [17]: The direct-call approach, e.g, used by InsDal [20], inserts probes which, upon execution, directly invoke the same method with a different argument, e.g., a string, to identify which basic block or statement was covered. However, since the regular *invoke* instruction can only reference the first 16 registers, it might be necessary to shift registers such that the arguments to the *invoke* instruction are stored within those first 16 registers.

To circumvent this problem, one can build the instrumentation upon the indirect-call approach, e.g., used by Huang et al. [17]. Here, the probe consists of the invocation of a distinct parameterless method for each location, which in turn then calls the general probing method with an argument to identify the covered basic

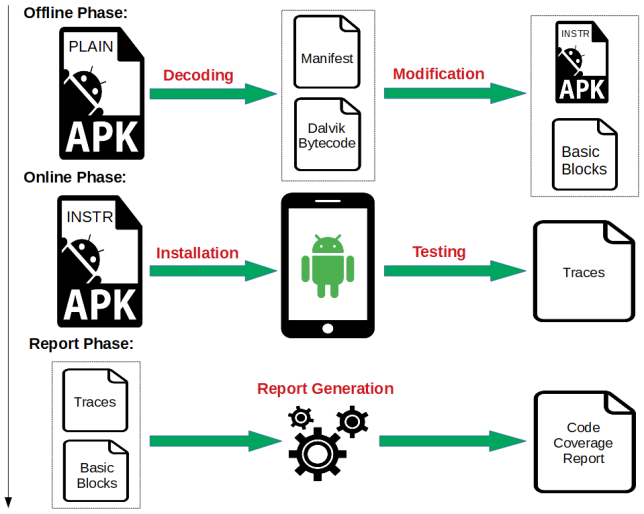

Figure 1: WallMauer's workflow.

block or statement. This avoids shifting registers at the original location but requires for each basic block or statement an additional method and imposes consequently a further overhead. Moreover, since each DEX file can only contain up to 65K methods, a split into multiple DEX files might be required.

Finally, an alternative approach implemented for instance by COSMO [24], instruments the Java bytecode rather than the Dalvik bytecode. This bypasses the above-mentioned restrictions, but the conversion process is not completely lossless and the re-translation may thus fail or lead to code different to the original one. For example, switch-tables are differently encoded by Dalvik and Java bytecode and therefore cannot be mapped 1:1.

3 DALVIK BYTECODE COVERAGE INSTRUMENTATION

The general workflow of the WallMauer approach consists of three sequential phases as illustrated by Fig. 1. First, in the offline phase (Section 3.1) the plain APK is decoded into the manifest and the Dalvik bytecode. Both artifacts are modified and repackaged into an instrumented APK. In addition, a file containing the instrumented basic blocks is produced, which is later required for the coverage computation. The instrumented APK is installed on a device during the online phase (Section 3.2) and explored either manually or in an automated fashion. Once the testing is completed, a file containing the visited basic blocks (traces) can be pulled from the device. Together with the file obtained during the instrumentation that contains the instrumented basic blocks a fine-grained code coverage report can be generated in the report phase (Section 3.3).

3.1 Offline Phase

In the offline phase the actual instrumentation takes place. Similar to ACVTool [22] and COSMO [24] we insert probes, but do so directly into the Dalvik bytecode and not the Smali representation or the Java bytecode, respectively.

Consider the instrumented method illustrated in Listing 2. The colored parts highlight the changes induced by the instrumentation. Our instrumentation requires two additional local registers, thus the number of registers in the first line is increased by two. To ensure that instructions can still address the parameter registers, we shift them to the left by two positions. This is highlighted by the first three instructions in the example (line 2–4). The *this*-reference originally stored in the register *p0* is moved to *v2*, while the two double values originally stored in *p1* and *p3* are moved to *v3* and *p1*, respectively. This shifting ensures that the original instructions still reference valid registers, and in addition the newly inserted registers are located at the end of the register stack, i.e., *p3* and *p4* can be freely used for our purposes.

The next two instructions (line 5–8) essentially represent our inserted probes at the beginning of each basic block. We load a string constant that uniquely identifies the given basic block into the free register *p3*. Afterwards, we invoke the static method *trace* of the *Tracer* class with the trace stored in the register *p3*. Note that the usage of the *invoke-static/range* instructions allows addressing up to 65536 registers in contrast to the regular *invoke-static* instruction, which is limited to the first 16 registers. The string constant used to identify a basic block (line 5–6) does not only describe the location of the basic block (first number) but also its size in terms of bytecode instructions (second number). This information allows us to actually compute line coverage over the bytecode instructions.

In most cases this rather simple probe is sufficient to track code coverage. However, there are certain edge cases that require a more comprehensive probe, consider for instance Listing 3. Assume that in the artificial example a new basic block is defined within the *try* block (line 4–10). In the regular case we would place our probe at the beginning of this block, i.e., at line 5 onwards. However, certain restrictions apply within *try-catch* blocks. In particular, the bytecode verifier checks at the beginning of each *catch* block (line 13) whether each register type is consistent throughout the *try* block, i.e., through all possible control flow paths the register must have the same type, otherwise the register state is said to be conflicted and the verifier would reject the code. Since the inserted *invoke-static/range* as part of our probe can theoretically throw an exception, an additional control flow path is introduced that may lead to a conflicted register state. To overcome this issue we use the same trick as ACVTool to bypass the verifier: We place our probe at the very end of the method (line 22–25) and insert a *goto* instruction at the beginning of the basic block (line 5) instead. The *goto* redirects the control flow to the end of the method (line 21), executes the probe and jumps back to the basic block (line 7) using another *goto* instruction (line 26). Since the bytecode verifier does not resolve such jumps it does not report a potentially conflicted register state, and the code can be successfully executed at runtime.

Another issue arises when a register state is conflicted at the beginning of a method. Such a register state is not forbidden per se, but one must re-initialize the register content before the register can be read (moved). This is accomplished by line 2 in Listing 3 before line 3 actually reads the value. We can use the value *0*, since this has the same internal representation for each data type.

A further restriction shown in the given example is that probes cannot be placed in front of *move-exception* instructions (line 14)

```
.method private calculateBMI(DD)D
.locals 4
move-object/from16 v2, p0
move-wide/from16 v3, p1
move-wide/from16 p1, p3
const-string p3, "Lcom/zola/bmi/BMIMain;
    ->calculateBMI(DD)D->0->7"
invoke-static/range {p3 .. p3},
    LTracer;->trace(Ljava/lang/String;)V
const-wide v0, 0x4001a305532617c2L
div-double/2addr v3, v0
const-wide v0, 0x3f9a027525460aa6L
mul-double p1, p1, v0
div-double/2addr v3, p1
div-double/2addr v3, p1
return-wide v3
.end method
```

Listing 2: Simple instrumentation probe.

```
.method private onChange(Z)V
move-object/from16 v3, p0
const/4 p1, 0x0
move/from16 v4, p1
:try_start_0
goto/32 :goto_0

:goto_1
invoke-direct v3, Lsl/ShoppingList;
    ->writeAllUnsavedChanges()V
:try_end_0
.catch Ljava/io/IOException;
    {:try_start_0 .. :try_end_0} :catch_0
:catch_0
move-exception v0
const-string p0, "Lsl/ShoppingList;
   ->onStop()V->38->7"
invoke-static/range {p0 .. p0},
    LTracer;->trace(Ljava/lang/String;)V
return-void

:goto_0
const-string p0, "Lsl/ShoppingList;
   ->onStop()V->36->2"
invoke-static/range {p0 .. p0},
    LTracer;->trace(Ljava/lang/String;)V
goto/32 :goto_1
.end method
```

Listing 3: Complex instrumentation probe.

although the *catch* block defines a new basic block. In this case, we place a probe directly afterwards (line 15–17).

The static *Tracer* class injected alongside with the probes is implemented as a broadcast receiver and is responsible for collecting the generated traces. Upon receiving a particular broadcast message, the collected traces are dumped to the devices' external storage. The class is registered in the manifest file alongside with the permission to write to the external storage. In case the permission is already present but only granted up to a specific API level, we remove the corresponding *android:maxSdkVersion* attribute from the permission tag. In order to avoid that traces are lost on unexpected app crashes, the *Tracer* class overwrites the uncaught exception handler,[1] which in turn dumps the collected traces upon a crash.

3.2 Online Phase

After the installation of the instrumented APK through the command line the app can be either explored manually or through an automated approach. In fact, any existing testing framework can be used for this purpose. As long as the exploration runs the *Tracer* class keeps collecting traces in the form of visited basic blocks (as injected during the instrumentation) in a set data structure, i.e., a trace looks identical to the string constant passed to the *trace* method. Once a pre-defined limit of traces is reached (currently set to 5000), the *Tracer* automatically dumps the collected traces to a file on the external storage. This intermediate dumping ensures that we do not overload the AUT while at the same time too frequent writes, e.g., after receiving a new trace, are avoided through the trace recording. Once the exploration is completed, we can append the remaining traces to the same file by invoking a broadcast with the action *STORE_TRACES* from the command line. If the AUT crashes during the exploration, the overridden exception handler by the *Tracer* class takes over and dumps the currently recorded traces to the same file. The online phase ends with pulling the traces file from the external storage through the command line.

[1] https://developer.android.com/reference/java/lang/Thread.UncaughtExceptionHandler

3.3 Report Phase

Together with the traces file from the online phase and the basic blocks file obtained during the offline phase, WallMauer can generate a code coverage report. Each trace records not only a visited basic block per method but also its size in terms of bytecode instructions, while the basic blocks file records the same information for all instrumented basic blocks. This allows us to compute line coverage at any level of abstraction, e.g., at method level. The output produced by WallMauer is currently a textual listing per class of the achieved line coverage, as well as the overall line coverage.

3.4 Architecture & Instrumentation Modes

The WallMauer tool is designed as a multi module Java command line application that consists of two parts: The instrumentation library that takes as input the plain APK and produces an instrumented APK alongside with the basic blocks file and the code coverage report generation library that takes the traces and basic blocks files to produce a fine-grained coverage report. Our implementation

builds largely upon the following three libraries: *Apktool*[2] to decode and repackage APKs, *dexlib2*[3] to modify the Dalvik bytecode and *multidexlib2*[4] to read from and write to multiple DEX files with automatic handling of *multidex* upon reaching the 65K method limit per DEX file.

WallMauer by default instruments at the basic block level to obtain line coverage but can be configured to instrument at method or branch level to report method or branch coverage, respectively. The instrumentation approach for these two modes is identical except that only a subset of probes needs to be inserted, thus having a lower overhead in general. Moreover, WallMauer can be configured to apply the instrumentation either to all classes, or, by default a lightweight mode is used in which only core application classes are instrumented. Here, we follow the typical convention that core application classes reside within the package or any subpackages described by the package name attribute listed in the manifest. Additionally, we whitelist the package or any subpackages the main activity resides in, and we exclude the untraceable resource class *R.java*[5] as well as the untraceable *BuildConfig.java*[6] class. This further reduces the overhead induced by the instrumentation, and is based on the observation that app testers are primarily interested in the coverage of the core application classes, rather than 3rd party libraries or Android-specific classes. Moreover, APKs often include classes or methods as part of dependencies that are not used by the application, effectively making them uncoverable.

3.5 Usage

The instrumentation of the APK happens by invoking *java -jar instrument.jar* with the respective APK that should be instrumented. (If you want to use the lightweight instrumentation mode append the flag *–only-aut.*) This will produce the instrumented APK and a file called *blocks.txt* that contains all instrumented basic blocks. This file is then later required for the coverage computation. After the instrumentation, one has to re-align and re-sign the APK. This can be accomplished by using the build tools *zipalign* and *apksigner* that come shipped with the Android SDK. Now, the instrumented APK should be installed on an emulator or a real device using for instance the command *adb install*. After that, the app can be explored manually or in an automated fashion. Once the exploration is completed, a broadcast needs to be sent to dump the collected traces to the external storage of the emulator or real device, respectively. This can accomplished by invoking *adb shell am broadcast -a STORE_TRACES*. Then, one can pull the traces to the local filesystem by calling *adb pull storage/emulated/0/traces.txt*. Together with previously acquired *blocks.txt* file the coverage report can be generated by invoking *java -jar coverage.jar blocks.txt traces.txt*.

3.6 Limitations

Since we unpack and repackage an APK during instrumentation, we need to re-sign it at the end. However, certain apps may employ some kind of anti-tampering techniques that in turn block modified APKs either completely or let them run only in a degraded mode. This limitation applies to all instrumentation tools. We rely upon the correct functioning of *apktool* to decompile and repackage an APK, which applies to all instrumentation tools that operate in a black-box manner, i.e., use the APK file as input. However, recent improvements of *apktool* have reduced related errors to a minimum, such that this limitation can be neglected. Due to some restrictions in the Dalvik bytecode, we can only instrument methods that use less than 255 registers in total. In particular, we insert two additional local registers and the usage of the *const-string* instruction to hold the trace does not allow specifying a register with an id greater than 255. However, this scenario applies to less than 0.1% in our experiment data set and is likely uncommon. Moreover, we could circumvent this limitation completely by using the indirect-call strategy, i.e., injecting a parameterless static method as probe, which in turn calls the *trace* method of our *Tracer* class. Since our tool can handle *multidex* out-of-the-box, we could have as many additional methods as we want. Ideally one would use the indirect-call strategy only for those methods that use more than 254 registers to keep the runtime overhead moderate.

[2]https://github.com/iBotPeaches/Apktool
[3]https://github.com/google/smali
[4]https://github.com/DexPatcher/multidexlib2
[5]https://developer.android.com/reference/android/R
[6]https://developer.android.com/reference/androidx/media3/database/BuildConfig

4 EVALUATION

We aim to answer the following research questions:

RQ1: How many apps can be successfully instrumented?
RQ2: What is the overhead induced by the instrumentation?
RQ3: Is the reported coverage compliant with other tools?

4.1 Study Subjects

Our experiments are based on a sample of 1000 open source apps. We leveraged the F-Droid XML index[7] that lists the available apps with metadata, e.g., version, min sdk version, etc., and randomly selected 1000 apps as follows: We first analyzed which apps are applicable on a x86 emulator image running API level 25 and do not represent games according to the category metadata tag, since MATE can only partially interact with screens of gaming apps. After that we performed a random sampling strategy during which we checked whether an app declares a launchable main activity in the manifest, which is a requirement for Monkey [13]. To ensure that the apps do not crash immediately, we then performed a sanity check by running each app for roughly one minute using Monkey. Apps that could not be installed or launched, or which crashed within the first minute, were excluded and replaced by a new sample. The resulting set of 1000 apps contains 306 *multidex* apps.

4.2 Instrumentation Tools & Environment

Since WallMauer is a black-box coverage instrumentation tool, we include ACVTool and COSMO as baselines in our empirical study. Both tools have proven to yield high success rates [22, 24], and they both report line coverage, allowing comparison with line coverage reported by WallMauer. Note that ACVTool can only partially instrument *multidex* apps, i.e., it only instruments the very first DEX file. We excluded tools that instrument at a different level of granularity, i.e., Ella [3], InsDal [20], CovDroid [28], the approach proposed by Huang et al. [17] and the prototype implementation given by Horváth et al. [15], since this would make the overhead

[7]https://f-droid.org/repo/index.xml

comparison (*RQ2*) unfair. Moreover, certain tool implementations are not publicly available, i.e., ABCA [18], while other tools like Asc [25], Androcov [21] or DroidFax rely upon *Jimple*—an alternative intermediate representation of Dalvik bytecode but less accurate than *Smali* [10], thus making the coverage compliance check non-trivial. On top of that, ACVTool has reported higher success rates for all previously mentioned tools that are publicly available. We do not consider white-box instrumentation tools in our empirical study, since we assume that we do not have access to the source code.

We used the latest available git commits for both ACVTool (HEAD@ce8e39c) and COSMO (HEAD@228f27e) and adapted both tools to use the latest available version of *apktool* (v2.9.0), since it is also used by WallMauer and can decode and re-package almost any APK in comparison to previous versions. In addition, we updated COSMO to use the latest available version of *dex2jar* (v2.4), which enhances COSMO with its support of *multidex*, and applied the patches discovered in a public fork[8] that integrated the missing configuration and library files and fixed the signing and aligning procedure after the instrumentation. With these improvements COSMO has the same preconditions as WallMauer for the instrumentation process. We configured WallMauer to instrument in lightweight mode at basic block level, which allows us to compare the reported line coverage to ACVTool. We also include a variant WallMauer* configured to instrument all classes within a DEX file, rather than the default lightweight mode that instruments only application classes, in order to have a fair comparison regarding the overhead (*RQ2*).

The instrumentation of the apps (part of *RQ1*) as well as verifying the compliance of the reported coverage (*RQ4*) was performed on a local workstation equipped with an Intel(R) Core(TM) i7-2600 CPU (8 cores) with 3.40 GHz and 16 GB of RAM, and runs Debian GNU/Linux 11 with Java 11. Experiments for checking the healthiness of apps (*RQ1*) and assessing the overhead (*RQ2*) were conducted on a compute cluster, where each node is equipped with two Intel Xeon E5-2620v4 CPUs (16 cores) with 2.10 GHz and 256 GB of RAM, and runs Debian GNU/Linux 11 with Java 11. We limit each execution to four cores and 60 GB of RAM, where the emulator (Nexus 5) runs a x86 image with API level 25 (Android 7.1.1) and is limited to 4 GB of RAM with a VM heap size of 576 MB. We make the implementation including the study subjects publicly available at https://figshare.com/articles/dataset/replication-package-instrumentation-study_zip/24438475.

4.3 Experimental Procedure

RQ1. In order to answer *RQ1* we instrumented the 1000 F-Droid apps using ACVTool, COSMO, Wallmauer and Wallmauer*, respectively. Any successfully generated instrumented APKs were explored with Monkey for roughly one minute. We consider an app to be successfully instrumented, i.e., healthy, if a trace or coverage file can be fetched from the emulator after the exploration and the respective coverage report can be generated.

RQ2. We measure the overhead induced by the instrumentation on the set of healthy apps, i.e., the set of 525 apps that could be successfully instrumented by all tools (*RQ1*), as follows: First, we compare the size of the DEX files before and after the instrumentation. This comparison was also performed in the *ACVTool* study [22]. Second, we also compare the instrumentation time overhead of every tool. Lastly, we want to quantify the runtime overhead. We originally thought that we could leverage Monkey using the same seed like was done in the *ACVTool* study [22], but after performing some preliminary experiments we noticed that specifying the same seed value does not guarantee that actually the same sequence of events is executed. This happens because Monkey can generate too many events in the fraction of a second for the Android emulator such that only an undefined number of events is actually received and executed. One could specify a large enough timeout between two events, but this in turn may compensate the overhead induced by the instrumentation. We tried the *PassMark* benchmark app also used in the *ACVTool* study [22], but there observed strange behaviour where the instrumented app performed better than the original app when instrumented with WallMauer or WallMauer*.

Thus, we decided to craft our own script that can automatically generate event sequences with a minimum length. The script launches the AUT and then randomly induces either a tap, swipe, text or key (e.g., volume up) event. This process is repeated until the specified length is reached. If any event causes a crash or the AUT is left, the app is reset and the process starts from the beginning. This ensures that only the AUT is explored without the need to restart the app in between. Once such a sequence is found the script outputs them in the form of *adb shell input* commands such that they can be easily replayed later on. In fact, our script is a simplified version of Monkey that operates at a speed such that all the events get definitely executed. It may take several iterations to produce such a sequence but it reduces disruptive factors that may compensate the runtime overhead, e.g., the time to check whether an event left the AUT and a restart in such scenario, to a minimum, since the produced event sequence is guaranteed to only explore the AUT without the need for intermediate restarts. We configured the script to produce a sequence of 60 events and this sequence was applied 10 times on both the plain and the instrumented app to mitigate disturbing effects caused through random load. We recorded the average execution time per app and computed thereout the percental runtime overhead.

In addition to each percental or absolute overhead we statistically compare each pair of tools using a Wilcoxon-Mann-Whitney U test at $\alpha = 0.05$, and the Vargha-Delaney $\hat{A}_{12}$ effect size [27]. An effect size $\hat{A}_{12} < 0.5$ between two tools implies that the first tool achieves lower values of the considered metric, while for an effect size $\hat{A}_{12} > 0.5$ the reverse holds.

RQ3. Since the *ACVTool* study [22] has shown that ACVTool is compliant with other state-of-the-art code coverage tools for Android and in particular compliant with JaCoCo, which is leveraged by COSMO under the hood, it is sufficient that we compare the reported coverage by WallMauer* to the reported coverage by ACVTool. Moreover, both tools report coverage based on Dalvik bytecode, simplifying the comparison. We took the 638 apps that could be successfully instrumented by both ACVTool and WallMauer* and explored them using Monkey with a deterministic sequence (same seed) of 1000 events. To ensure that Monkey is

[8] https://github.com/ggiraldez/COSMO – HEAD@c8af5a3

not sending too many events within a second (recall that this can lead to dropping events) we specified a delay of 300ms between two events. We repeated this execution three times per app and tool and recorded the average coverage. In addition, we randomly selected 10 apps and performed manual testing, i.e., we applied for each app the same sequence of events with both instrumented versions and compared the reported coverage. This should ensure that non-observable coverage differences during the former experiment are not caused by the dynamic nature of certain apps.

4.4 Threats to Validity

Threats to external validity may arise from our sample of apps, and results may not generalize beyond the tested apps. To counteract selection bias, we picked 1000 apps for the evaluation randomly, but apps on *F-Droid* may be different from those on *Google PlayStore*. For consistency we used one specific emulator configuration and one concrete API level, but results may differ on other versions.

Threats to internal validity may arise from bugs in our instrumentation tool or our analysis scripts. To mitigate this risk, we manually reviewed the results, and tested and reviewed all code. To reduce the risk of favoring one instrumentation tool over the other, we equipped all tools with the same external dependencies, e.g., the same build tools for signing and aligning APKs and the same version of *apktool*. The patches we applied to the COSMO implementation have been carefully reviewed. Lastly, to allow for a fair comparison of the overhead and the reported coverage values, we instrumented the same classes in each APK.

Threats to construct validity may result from our choice of metrics, in particular for overhead. However, we relied upon metrics also applied by other researchers.

4.5 RQ1 Results

Figure 2 summarizes for how many apps an instrumented APK was produced, and how many apps remained healthy after the instrumentation. ACVTool produced an instrumented APK in 885 out of 1000 cases, of which 638 remained healthy, while COSMO produced 891 instrumented APKs, of which 793 remained healthy. In contrast, both WallMauer variants instrumented all but one app, and 998 remained healthy with WallMauer and 996 with WallMauer*. This clearly demonstrates the robustness of our approach.

To better understand the reasons why instrumentation failed in some cases, we studied the logs produced by each tool and Monkey. ACVTool failed to produce an instrumented APK in 115 cases. Of these, 64 cases can be directly attributed to the *multidex* problem. In 49 cases the Dalvik bytecode was corrupted either due to an invalid register assignment or a jump label address out of the eligible range. The remaining two failures are caused by *apktool* when reassembling the APK. One of those two affects all tools, while one failure is specific to ACVTool. We believe that this issue is caused by ACVTool invoking *apktool* with the *aapt2* binary provided from the Android SDK and not the bundled one from *apktool*, which comes with a few patches.

In total, 247 apps (885–638) did not remain healthy after instrumentation with ACVTool. In 162 cases the crash originated from the code that was injected by the tool. In 29 cases no coverage file was generated, which in four cases could be directly attributed

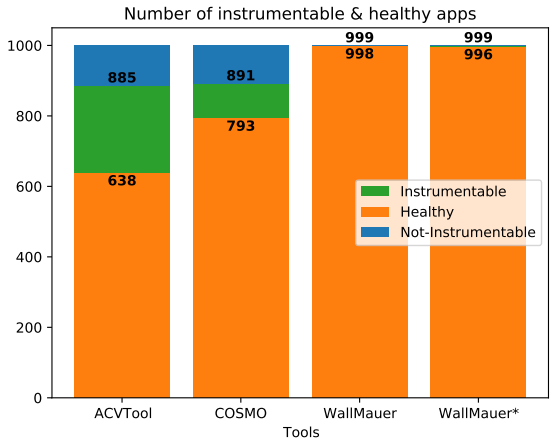

Figure 2: Number of instrumentable and healthy apps for ACVTool, COSMO, WallMauer and WallMauer*.

to the application not responding, while in two cases there were corrupted coverage files. In the remaining 54 cases the app was not installable anymore, due to a fault in the aligning procedure. In particular, the *zipalign* binary was missing a special flag[9] that is required with more recent API levels. Since ACVTool only instruments the first DEX file in an APK, a subset of the apps is only partially instrumented. Of the 306 *multidex* apps ACVTool managed to partially instrument only 139 apps successfully.

COSMO failed to produce an instrumented APK in 109 cases, of which 102 can be attributed to a failure in the Java to Dalvik bytecode conversion. Although *dex2jar* produced a JAR file, it is likely that something was corrupted during this process, which then broke the re-conversion. Two failures appeared during the offline instrumentation using JaCoCo and one failure is due to the *apktool* building issue previously mentioned. In total, 98 apps (891–793) did not remain healthy after instrumentation with COSMO. In all of these cases no coverage file was produced. In 67 of these cases the problem can be attributed to the application not responding.

Both WallMauer variants failed to produce an instrumented APK in only one case, which is due to the *apktool* issue affecting all tools. The one app (999–998) as well as the three apps (999–996) that did not remain healthy after the instrumentation with WallMauer and WallMauer*, respectively, exhibit the same fault: The instrumentation caused too much load on the main thread of the apps and thus those apps got stuck upon launching, and consequently no traces file were produced. We verified that the one app (WallMauer) and the three apps (WallMauer*) can actually be instrumented and are not stuck due to any anti-tampering technique employed by the apps by only instrumenting the main activity, which allowed the app to be launched and explored by Monkey successfully. This undermines that the actual instrumentation is not causing any verification errors in the produced Dalvik bytecode, which is often a primary reason why other tools fail.

[9] https://stackoverflow.com/questions/55173004/targeting-sdk-android-q-results-in-failed-to-finalize-session-install-failed-i

We will inform the tool authors of ACVTool and COSMO with our insights and provide patches where applicable.

Summary (*RQ1*): WallMauer instrumented 998 out of 1000 F-Droid apps successfully, which demonstrates its improved robustness compared to the state-of-the-art tools COSMO (793) and ACVTool (638).

4.6 RQ2 Results

4.6.1 Instrumentation Time. There are 525 apps that were successfully instrumented by *all* tools. The instrumentation time overhead on these apps is shown in Fig. 3. On average, it takes 71 seconds to instrument an app using ACVTool, while COSMO accomplishes the same task in 35 seconds. In contrast, WallMauer requires 21 seconds and WallMauer* 99 seconds. There is a significant difference between WallMauer and ACVTool ($p < 0.001$, $\hat{A}_{12} = 0.11$) in favor of WallMauer, and between WallMauer and COSMO ($p < 0.001$, $\hat{A}_{12} = 0.24$).

Note that the instrumentation time for ACVTool were higher if it would instrument *all* DEX files in multidex APKs, rather than just the first one. Out of the 525 apps instrumented by all tools there are exactly 100 *multidex* apps (19.05%); on these apps ACVTool is likely to produce incomplete coverage reports as a consequence of the incomplete instrumentation.

COSMO is extremely fast in comparison to WallMauer* and ACVTool. We believe that this can be attributed to the fact that COSMO can perform some operations directly in place and that it does not need to generate any additional files for the coverage computation unlike the other tools.

The large performance difference between WallMauer and WallMauer* is exacerbated by the large number of multidex apps. Nevertheless, we were a bit surprised that WallMauer* performs rather slow although it instruments all classes and methods in a parallel manner. We identified as root cause that the synchronized writes to the *blocks.txt* are hampering the performance as well as the rather slow read and write operations induced by the underlying *multidexlib2* library. Furthermore, we need to perform some static analysis like the identification of the leader instructions to form the basic blocks or the register type inference algorithm that requires some computation time.

Overall, we nevertheless believe that the current instrumentation time overhead of WallMauer* is tolerable for instrumenting all classes, but for most practical purposes the lightweight instrumentation provided by WallMauer is sufficient, and very fast.

4.6.2 Size. Another common metric to quantify the instrumentation overhead is to measure the percental size increase of the DEX files [22]. This information is represented in Fig. 4. ACVTool imposes a size overhead of 177% on average, while COSMO causes an average increase of 125% (with a median of 41%, but a few extreme outliers). In contrast, WallMauer induces a small overhead of 35%, while WallMauer* triples the size (313%). There is a statistically significant difference between WallMauer and ACVTool ($p < 0.001$) in favor of WallMauer ($\hat{A}_{12} = 0.02$), while the same holds between WallMauer and COSMO ($p < 0.001$, $\hat{A}_{12} = 0.22$).

We can only speculate why COSMO imposes 52% less overhead compared to ACVTool although they use the same representation.

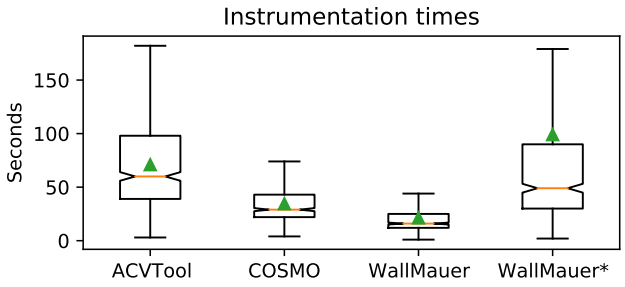

Figure 3: Instrumentation time distributions.

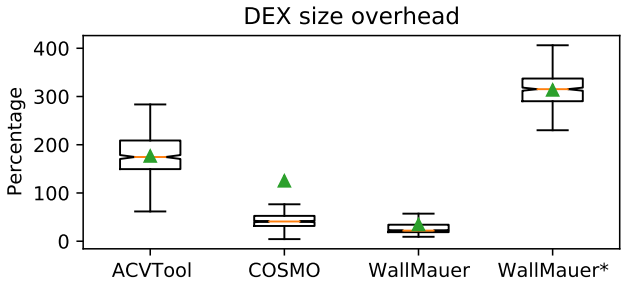

Figure 4: Percental DEX size increase.

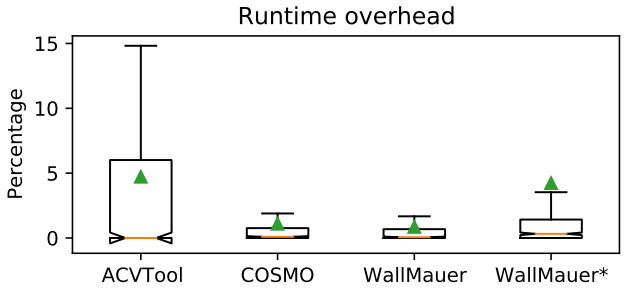

Figure 5: Percental runtime overhead.

One reason might be that during the conversion from Dalvik to Java bytecode or vice versa some optimizations may take place.

The larger overhead of WallMauer* can be attributed to the fact that we use a verbose string to uniquely identify different basic blocks, while ACVTool and COSMO rely upon a compact representation of Boolean vectors. However, the size overhead caused by the strings does not constitute a problem as long as the instrumented apps remain operational and do not exhibit a significant runtime overhead. In *RQ1* (Section 4.5) we observed that three apps were affected by too much load imposed through the instrumentation of WallMauer* and thus were unresponsive, while ACVTool (four cases) and COSMO (67 cases) performed worse in this regard. By being selective about instrumentation, WallMauer results in smaller DEX files despite the use of string constants.

4.6.3 Runtime. Of particular interest in terms of overhead are effects on the runtime. For this analysis we had to exclude 27 apps in total because our script could not produce a sequence consisting

of 60 inputs without leaving the AUT in a reasonable amount of time. This primarily happens because certain apps directly open some system app after the startup. The crafted test, however, intends to measure the overhead exclusively on the AUT.

Figure 5 shows the percental runtime overhead on a set of 498 apps (525 - 27). ACVTool imposes an average runtime overhead of 4.72%, while COSMO induces four times less overhead with 1.09%. WallMauer exhibits the lowest runtime overhead with 0.85% on average and WallMauer* comes with an average overhead of 4.23%. We argue that such overheads are tolerable for a fine-grained coverage instrumentation. There is a significant difference between WallMauer and ACVTool ($p = 0.04$) with a negligible effect size ($\hat{A}_{12} = 0.47$) in favor of WallMauer, while between WallMauer and COSMO there is no significant difference ($p = 0.32$).

Summary (*RQ2*): WallMauer improves over ACVTool and COSMO with an average of 21 seconds to instrument an app, increasing the DEX size by 35%, and a runtime overhead of 0.85%.

4.7 RQ3 Results

Figure 6 shows the coverage box plots for both ACVTool and WallMauer* on a set of 633 apps. We had to exclude five apps from the original set of 638 apps successfully instrumented by ACVTool because it failed to reliably retrieve the coverage in those cases. One can hardly observe any visible differences in Fig. 6, and a high p-value of 0.76 reinforce the absence of significant differences; ACVTool achieves a mean coverage of 12.97% while WallMauer* reaches 13.18% on average. Moreover, in only 18 cases there was a tool-wise difference greater than 5%, which is primarily due to the fact that ACVTool does instrument only the first DEX file, while in 461 cases the deviation was less than 1%. A similar observation can be made by inspecting Table 1 which shows the line coverage reported by ACVTool and WallMauer* on the 10 selected apps, based on manual testing to ensure identical execution sequences. Overall, Table 1 shows large compliance between WallMauer* and ACVTool, and thus confirms that the approach implemented in WallMauer* provides accurate coverage values.

There consistently are minor discrepancies of less than 1% per app. These are caused by ACVTool considering certain instruction as untraceable, in particular control-flow changing instructions like *return*, *goto* and *throw*, whereas WallMauer* succeeds in tracing these instructions as well. We also observed that *invoke* instructions that are paired with *move-result* instructions are also not traced by ACVTool. This is a direct limitation of the instruction-based instrumentation; recall that it is not allowed to place a probe between such a pair (cf. Section 2.5). Lastly, no probes can be placed between a *catch* label and the *move-exception* instruction. In contrast, WallMauer* can track all of these instructions by placing the probe at the beginning of a basic block. The only exception applies to the latter restriction, here we place our probe directly after the *move-exception* instruction.

Summary (*RQ3*): The reported coverage between ACVTool and WallMauer* is widely compliant, except for minor discrepancies caused by limitations of ACVTool.

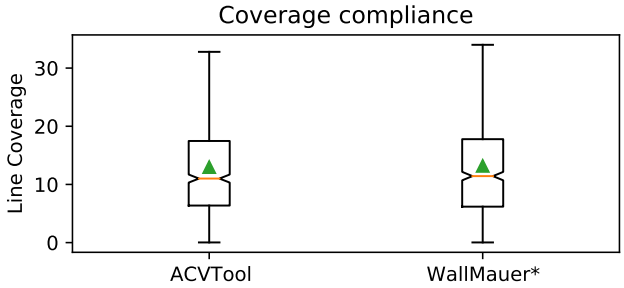

Figure 6: Coverage compliance between ACVTool and WallMauer*.

Table 1: Line coverage compliance check.

Total line coverage on the selected 10 apps.

App	Line Coverage		Deviation
	ACVTool	WallMauer*	
at.linuxtage.companion	18.12%	18.59%	0.47%
br.com.colman.nato	36.75%	37.63%	0.88%
org.bibledit.android	3.59%	3.66%	0.07%
org.yaaic	0.77%	0.78%	0.01%
taco.scoop	16.11%	16.64%	0.53%
uk.co.yahoo.p1rpp.calendartrigger	2.03%	2.31%	0.28%
x1125io.initdlight	7.02%	6.87%	0.15%
org.michaelevans.nightmodeenabler	5.26%	5.42%	0.16%
net.ibbaa.keepitup	12.17%	12.46%	0.29%
eu.faircode.netguard	24.64%	25.23%	0.59%

5 CONCLUSIONS

In this paper we proposed a refined approach to measure code coverage in Android apps, and its implementation in the WallMauer tool. WallMauer requires solely the APK file as input and thus can be relatively simply integrated into any Android testing frameworks. In contrast to most state-of-the-art tools like ACVTool it can handle *multidex* out-of-the-box and strictly adheres to the Dalvik bytecode specification, making it very robust against verification errors. Empirical results on a set of 1000 F-Droid apps demonstrated that WallMauer is extremely robust, successfully instrumenting 99% of the apps.

There nevertheless are possibilities for future improvement. The current code coverage report is textual but we would like to provide a graphical report, to better support human testers. Another aspect is the limitation that methods using more than 254 registers cannot be instrumented so far. As already indicated, this limitation can be circumvented using the indirect-call approach. We believe that we can further reduce the size and runtime overhead by using a more compact representation for the basic blocks, e.g., a Boolean array.

WallMauer is available as open source, and we provide a replication package that contains all case studies, the tool implementation, and the raw results of our study at: https://figshare.com/articles/dataset/replication-package-instrumentation-study_zip/24438475.

ACKNOWLEDGMENTS

This work is supported by DFG project FR2955/4-1 "STUNT: Improving Software Testing Using Novelty".

REFERENCES

[1] [n. d.]. App components. https://developer.android.com/guide/components/fundamentals#Components. Accessed: 2023-10-10.

[2] [n. d.]. Dalvik bytecode. https://source.android.com/docs/core/runtime/dalvik-bytecode. Accessed: 2023-10-10.

[3] [n. d.]. ELLA: A Tool for Binary Instrumentation of Android Apps. https://github.com/saswatanand/ella. Accessed: 2023-10-10.

[4] [n. d.]. EMMA: a free Java code coverage tool. https://emma.sourceforge.net/. Accessed: 2023-10-10.

[5] [n. d.]. Enable multidex for apps with over 64K methods. https://developer.android.com/build/multidex. Accessed: 2023-10-10.

[6] [n. d.]. IntelliJ IDEA code coverage runner. https://www.jetbrains.com/help/idea/2017.1/code-coverage.html. Accessed: 2023-10-10.

[7] [n. d.]. JaCoCo Java Code Coverage Library. https://www.eclemma.org/jacoco/. Accessed: 2023-10-10.

[8] [n. d.]. Understand the APK structure. https://developer.android.com/topic/performance/reduce-apk-size#apk-structure. Accessed: 2023-10-10.

[9] Domenico Amalfitano, Nicola Amatucci, Anna Fasolino, and Porfirio Tramontana. 2015. AGRippin: A Novel Search Based Testing Technique for Android Applications. 5–12. https://doi.org/10.1145/2804345.2804348

[10] Yauhen Leanidavich Arnatovich, Hee Beng Kuan Tan, Sun Ding, Kaiping Liu, and Lwin Khin Shar. 2014. Empirical Comparison of Intermediate Representations for Android Applications. In *International Conference on Software Engineering and Knowledge Engineering*. https://api.semanticscholar.org/CorpusID:16483716

[11] Michael Auer, Felix Adler, and Gordon Fraser. 2022. Improving Search-Based Android Test Generation Using Surrogate Models. In *Search-Based Software Engineering*, Mike Papadakis and Silvia Regina Vergilio (Eds.). Springer International Publishing, Cham, 51–66.

[12] Michael Auer, Andreas Stahlbauer, and Gordon Fraser. 2023. Android Fuzzing: Balancing User-Inputs and Intents. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 37–48. https://doi.org/10.1109/ICST57152.2023.00013

[13] Android Developers Docs. [n. d.]. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey. Accessed: 2023-10-10.

[14] Marcelo Medeiros Eler, José Miguel Rojas, Yan Ge, and Gordon Fraser. 2018. Automated Accessibility Testing of Mobile Apps. In *ICST*. IEEE, 116–126.

[15] Ferenc Horváth, Szabolcs Bognár, Tamás Gergely, Róbert Rácz, Árpad Beszédes, and Vladimir Marinkovic. 2014. Code Coverage Measurement Framework for Android Devices. *Acta Cybern.* 21, 3 (aug 2014), 439–458. https://doi.org/10.14232/actacyb.21.3.2014.10

[16] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergo? Balogh, and Tibor Gyimóthy. 2019. Code Coverage Differences of Java Bytecode and Source Code Instrumentation Tools. *Software Quality Journal* 27, 1 (mar 2019), 79–123. https://doi.org/10.1007/s11219-017-9389-z

[17] Chun-Ying Huang, Ching-Hsiang Chiu, Chih-Hung Lin, and Han-Wei Tzeng. 2015. Code Coverage Measurement for Android Dynamic Analysis Tools. In *2015 IEEE International Conference on Mobile Services*. 209–216. https://doi.org/10.1109/MobServ.2015.38

[18] Shang-Yi Huang, Chia-Hao Yeh, Farn Wang, and Chung-Hao Huang. 2015. ABCA: Android Black-box Coverage Analyzer of mobile app without source code. In *2015 IEEE International Conference on Progress in Informatics and Computing (PIC)*. 399–403. https://doi.org/10.1109/PIC.2015.7489877

[19] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. 2013. Deriving code coverage information from profiling data recorded for a trace-based just-in-time compiler. *ACM International Conference Proceeding Series*, 1–12. https://doi.org/10.1145/2500828.2500829

[20] Jierui Liu, Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. 2017. InsDal: A safe and extensible instrumentation tool on Dalvik byte-code for Android applications. 502–506. https://doi.org/10.1109/SANER.2017.7884662

[21] Nataniel P. Borges, Maria Gómez, and Andreas Zeller. 2018. Guiding App Testing with Mined Interaction Models. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 133–143.

[22] Aleksandr Pilgun, Olga Gadyatskaya, Yury Zhauniarovich, Stanislav Dashevskyi, Artsiom Kushniarou, and Sjouke Mauw. 2020. Fine-Grained Code Coverage Measurement in Automated Black-Box Android Testing. *ACM Trans. Softw. Eng. Methodol.* 29, 4, Article 23 (jul 2020), 35 pages. https://doi.org/10.1145/3395042

[23] Anshuman Rohella and Shingo Takada. 2018. Testing Android Applications Using Multi-Objective Evolutionary Algorithms with a Stopping Criteria. 308–353. https://doi.org/10.18293/SEKE2018-084

[24] Andrea Romdhana, Mariano Ceccato, Gabriel Claudiu Georgiu, Alessio Merlo, and Paolo Tonella. 2021. COSMO: Code Coverage Made Easier for Android. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 417–423. https://doi.org/10.1109/ICST49551.2021.00053

[25] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI testing for Android applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 27–37. https://doi.org/10.1109/ASE.2017.8115615

[26] Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. 2016. Negative Effects of Bytecode Instrumentation on Java Source Code Coverage. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 225–235. https://doi.org/10.1109/SANER.2016.61

[27] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[28] Chao-Chun Yeh and Shih-Kun Huang. 2015. CovDroid: A Black-Box Testing Coverage System for Android. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, Vol. 3. 447–452. https://doi.org/10.1109/COMPSAC.2015.125

[29] Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, and Yutian Tang. 2023. Selectively Combining Multiple Coverage Goals in Search-Based Unit Test Generation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 91, 12 pages. https://doi.org/10.1145/3551349.3556902