Universiteit Leiden
The Netherlands

Opleiding Informatica

Composing instrumentation tools for Android apps

Arthur van der Staaij

Supervisors:
Olga Gadyatskaya & Nathan D. Schiele

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

July 23, 2021

Abstract

Many apps are developed for the mobile operating system Android every day, and they are used by an ever increasing amount of people. This has given rise to a growing need to analyse these apps, most importantly for security purposes. Many Android app analysis tools use a technique called *dynamic analysis*, which involves executing apps and gathering information while they are running. These analysis tools often have no access to the source code of the application under analysis: they have to make do with the compiled Android package. Therefore, many dynamic analysis tools for Android involve *bytecode instrumentation*: the modification of the compiled app code. Such instrumentation tools have all kinds of purposes, ranging from the measurement of code coverage to malware detection. Given this variety, it may be useful to work with multiple tools that use instrumentation at the same time. The composition of such tools may however lead to all kinds of problems, since their changes to the applications under analysis may conflict with each other. To facilitate the composition of multiple instrumentation tools, we propose a two-step approach involving *instrumentation blueprints*, reports of the instrumentation changes a tool needs to apply. We have designed a prototype syntax for these blueprints, adapted a modern instrumentation tool to emit them and implemented a prototype blueprint application program. Although there are still some limitations, our evaluation shows that our proposed approach can indeed work in practice.

Contents

1 Introduction

Over the last few years, the market of mobile phones has continued to grow. According to [1], approximately 6.648 million people use a smartphone as of 2021, and this number is expected to grow even further. The most common mobile operating system is Android, with a market share of around 72.8% [2, 3]. Many apps are developed for the operating system every day: in February 2021, a total of approximately 88 500 apps were released through Google's app store, and that was a relatively slow month [4]. As the prevalence of Android increases, so does the demand for tools that analyze its apps. And indeed, many such tools are available.

There are various ways in which Android apps can be analyzed. The techniques can broadly be split into two categories: *static analysis* and *dynamic analysis*. The former is based on analyzing an Android package "from the outside", without running it. The latter, on the other hand, involves executing the application in some form and gathering information from its behavior. Of course, tools may also use a combination of static and dynamic analysis. Analysis tools can also be categorized into *white-box* and *black-box* tools. White-box tools require access to the source code of an application, whereas black-box tools can perform their analysis using only the compiled application package. This thesis concerns black-box dynamic analysis tools, tools that involve running applications and which do not require source code access.

Black-box dynamic analysis tools often use a technique called *instrumentation*: they modify the (compiled) code of the application in order to gather information while it is running.

Even though a multitude of systems and frameworks related to instrumentation is continually being developed, a topic that has not been considered in detail is the *composition* of instrumentation tools: using multiple such tools at once. One can expect various issues to arise when combining multiple instrumentation tools, as they are generally not designed with other tools in mind.

In order to facilitate the composition of instrumentation tools, this thesis introduces the concept of *instrumentation blueprints*: specifications of the instrumentation changes applied by a tool. Instead of instrumenting applications directly, tools can output a blueprint, and an applicator system can subsequently apply multiple such blueprints at once. Because the applicator has knowledge of all the required code changes of the different tools at once, it can avoid issues that would otherwise be caused by the composition of the tools.

Besides introducing the concept of instrumentation blueprints and providing a basic syntax for them, this thesis also includes the implementation of a tool that can apply them and a practical implementation of blueprint output for ACVTool, a recent black-box code coverage tool that employs instrumentation [5].

To summarize, the contributions of this work are:

1. The design of an approach for the composition of instrumentation tools, based on instrumentation blueprints.
2. The definition of a prototype syntax for instrumentation blueprints.
3. The implementation of blueprint output for ACVTool.
4. The implementation of a prototype blueprint applicator program.

2 Background

This section contains background information that is necessary to understand the subject and contributions of this work. To begin, we will describe some internals of the Android operating system and the structure of Android applications. We will then move to the current approaches to Android application analysis. After that, we will go into more detail on the format of the code to which Android applications are compiled, and how and why this code is instrumented. Finally, we will give some background on the main topic of this thesis: the composition of instrumentation tools.

2.1 Android

We will first describe the basics of the environment in which Android applications are executed: the Android operating system. Android consists of multiple layers, which are commonly referred to as the Android stack. Figure 1 on the following page gives an overview.

At its core, Android is based on the Linux kernel, but there are many abstractions between it and the applications that the operating system supports. Of particular interest is the Android Runtime (ART), a virtual machine similar to the Java Virtual Machine: it executes code that has been compiled to low-level, machine code-like bytecode. The bytecode supported by the Android Runtime is called Dalvik, a remnant of the older Dalvik Virtual Machine that Android used before. Android applications are typically written in a high-level language such as Java, and then compiled to bytecode, which is run on the Android Runtime. They may however include native machine code as well, which needs to be compiled for the various different kinds of underlying hardware that phones may use. This work deals heavily with Dalvik bytecode, so we will come back to it further down.

On top of — among other components — the Android Runtime, a Java API framework is available to develop applications against. Android applications notably do not have a single defined entry point. Instead, they consist of multiple "app components" like UI screens, which are classes from the API framework. The Android operating system activates these components based on user interactions. Applications may also invoke components of other apps through a system called inter-component communication (ICC). For example, an app that requires a photo can send a message requesting a camera app to open and return the photo that was taken. This architecture is what gives the Android operating system its interconnectivity [7].

2.2 Application structure

Applications for Android come in the form of an Android Package, or APK for short. An APK contains all components that make up an application, such as Dalvik bytecode, native code and assets like images and XML files. They also include a manifest file, which is an XML file that holds the name of the package, its components, required permissions and other metadata.

Packages must also be given a cryptographic signature before they can be installed. This signature can be used to verify whether different packages have been created by the same developer. This is important for the secure delivery of updates, for example.

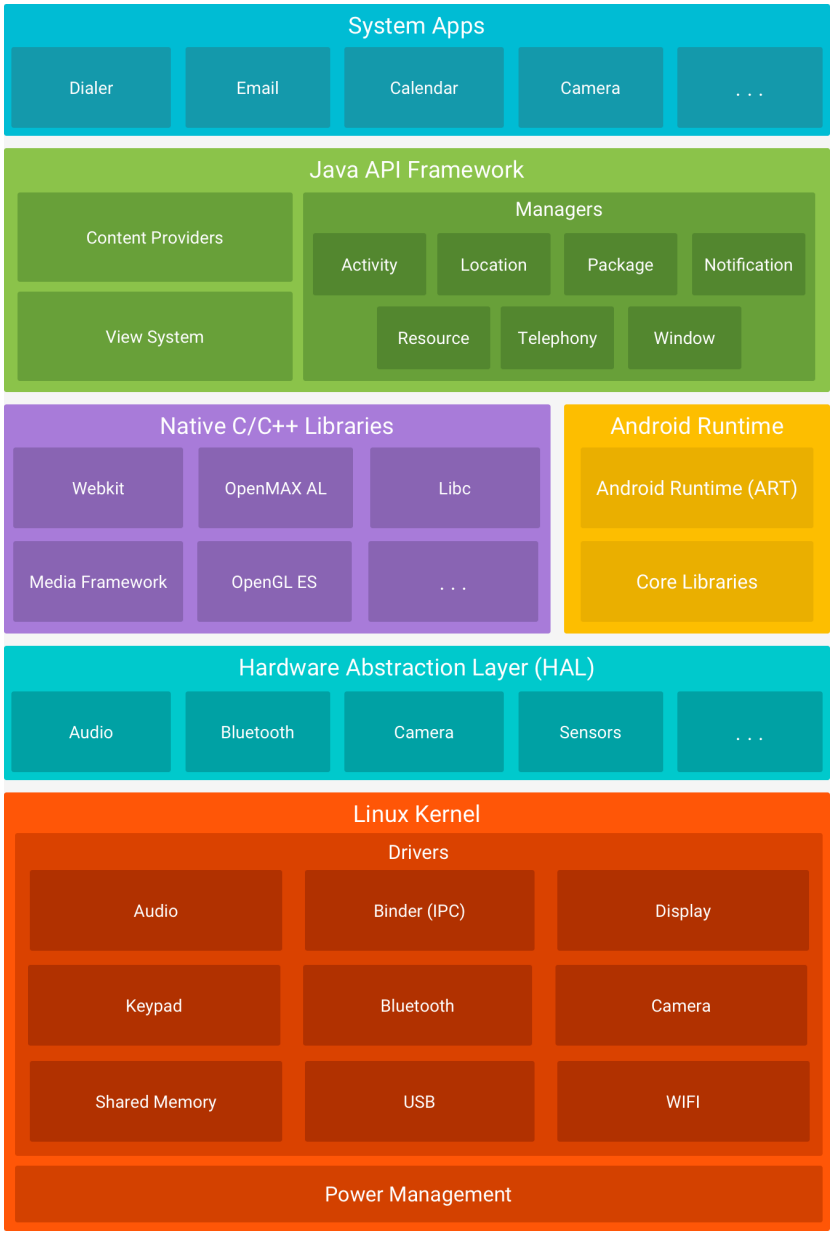


Figure 1: The Android Stack [6]

2.3 Android application analysis

Many tools are available for the analysis of Android applications. The purposes of these tools range from performance measurements to automated testing. Possibly the most important reason to analyse applications, however, is security. The widespread use of Android has made it an attractive target for malware authors. Phones often contain a wealth of information about their user, such as contacts and location, so the stakes can be high. Because we cannot always establish trust in the developers of Android applications, there is a need to be able to analyze these apps. New analysis tools continue to be developed to detect new kinds malware, and they use a variety of techniques to do so.

As touched upon in the introduction, there are multiple ways in which Android analysis tools can be categorized. Firstly, tools generally perform either *static analysis* or *dynamic analysis*, although hybrids do exist. Orthogonally to this, it is possible to distinguish between *white-box* and *black-box* tools. The contributions of this thesis apply specifically to black-box dynamic analysis tools. We will now discuss each categorization in a bit more detail.

White-box tools require the original source code of an application, usually written in Java. Black-box tools, on the other hand, can analyze compiled application packages without additional information: they operate on the level of Dalvik bytecode. White-box tools may be able to provide more information than black-box tools do, but they can generally only be used by the developers of the analyzed app. For other cases, black-box analysis is often the only option. Any tool that is meant to analyze apps from other parties, like security analysis tools, must be black-box.

Static analysis involves the examination of an application from an outside perspective. In contrast, dynamic analysis is based on executing an application in some way (usually in an isolated environment) and observing how it behaves in practice. Black-box static analysis tools generally look at an application's manifest file and analyze patterns and information flows in their code. On the other hand, black-box dynamic analysis tools rely on techniques like system call monitoring and bytecode instrumentation (the main subject of this work).

One might wonder why tools would resort to dynamic analysis instead of static analysis. After all, static analysis provides the certainty that apps are analyzed fully, something that is not guaranteed for dynamic analysis. There are multiple reasons for this. One is that some metrics, such as performance, can only be accurately determined dynamically. Another reason, one that is of particular importance for security analysis, is that many kinds of behavior are not easily captured with static analysis alone. Examples include behavior that depends on user input, dynamic loading of code [8, 9, 10], and inter-component communication [11, 12]. The use of advanced code obfuscation, involving techniques like Java reflection or bytecode encryption, can also prevent static analysis tools from fully understanding an app's behavior [13].

The primary trade-off of dynamic analysis compared to static analysis is that it can only detect behaviors that are actually exhibited by an app while it is being analyzed. In fact, there are some known techniques that malicious apps can use to exploit this weakness. They may not run any malicious code unless some highly specific conditions are met (a *logic bomb*) [14], or they may detect whether they are running in an emulated environment and not execute any malicious code if this is the case [15]. Since neither static nor dynamic analysis is perfect, both analysis styles are used in practice.

2.4 Dalvik bytecode and the Smali representation

Dalvik bytecode is in many ways similar to real machine code, consisting of low-level instructions like `add` and `goto`. There are however some significant differences as well. First of all, many high-level concepts of the source code (usually Java) remain visible in the bytecode. These include things like classes, fields, methods and types. Second, the Android Runtime is a

register-based virtual machine. This means that there are no memory access instructions in the bytecode, and there is no stack. Instead, functions have parameter registers and declare the amount of *local registers* they need. A total number of 65 536 registers is supported, far more than most real-world machines have.

Because Dalvik bytecode is a machine-like language of zeroes and ones, we don't deal with it directly. Instead, we use a human-readable assembly-like representation of it. There are two such representations that are commonly used: Smali [16] and Jimple [17]. Smali, the output of Gruver's `smali/baksmali` tool, has been designed specifically for Dalvik bytecode and stays very close to it. Jimple is a bit more abstract, and was primarily created for Java bytecode. This work uses the Smali representation.

We use Apktool [18], which relies on `smali/baksmali`, to disassemble APKs into smali files. A separate file is used for every Java class. Figure 2 on the next page shows an example of a Java class and its Smali representation. For clarity, we have removed some debug information and optional reflection metadata from the Smali code. Note how the class `Foo`, its methods `bar` and `baz` (and its implicit constructor) and its field `value` can all still be identified. The lines that start with a dot are called *directives*.

For each method in the original Java code, there is a corresponding `.method`/`.end method` block. Such a block begins with a header containing the method descriptor: the name of the method, its parameters and its return type. For example, the descriptor of `bar` is `bar(I)V`, as it requires one parameter of type `int` (`I`), and its return type is `void` (`V`).

The first line after the method header declares how many local register the method uses. All methods in the example use one. Instead of using `.locals`, as in the example, methods can also use `.registers` to specify the *total* number of registers (local and parameter). Inside a function, local registers are referenced with `v<number>`. The registers containing the method's parameters use the `p` prefix instead. All non-`static` methods also have an implicit *this*-parameter, which is placed in `p0`.

Although local and parameter registers use different names, there is actually no distinction: parameters are simply placed in the last registers of the method. If a method has `n` local registers, the first parameter register is `vn`. The `p`-names refer to the exact same registers; they are merely aliases. Table 1 shows the relation between the `v`- and `p`-registers for a non-static method with two local registers and three parameters (including the *this*-parameter).

The rest of the method body consists mostly of instructions, which are marked with red. Those who are familiar with assembly languages may recognize some similarities. The full instruction list can be found at [19].

`v0`	First local register
`v1`	Second local register
`v2 = p0`	First parameter register (*this*)
`v3 = p1`	Second parameter register
`v4 = p2`	Third parameter register

Table 1: Example register layout (adapted from [20])

```java
package com.example;

public class Foo {
  private void bar(int count) {
    for(int i = 0; i < count; i++) {
      baz(i);
    }
  }

  private void baz(int i) {
    value += i;
  }

  private int value = 0;
}
```

(a) Java

```
.class public Lcom/example/Foo;
.super Ljava/lang/Object;
.source "Foo.java"

# instance fields
.field private value:I

# direct methods
.method public constructor <init>()V
  .locals 1
  invoke-direct {p0}, Ljava/lang/Object;-><init>()V
  const/4 v0, 0x0
  iput v0, p0, Lcom/example/Foo;->value:I
  return-void
.end method

.method private bar(I)V
  .locals 1
  const/4 v0, 0x0
  :goto_0
  if-ge v0, p1, :cond_0
  invoke-direct {p0, v0}, Lcom/example/Foo;->baz(I)V
  add-int/lit8 v0, v0, 0x1
  goto :goto_0
  :cond_0
  return-void
.end method

.method private baz(I)V
  .locals 1
  iget v0, p0, Lcom/example/Foo;->value:I
  add-int/2addr v0, p1
  iput v0, p0, Lcom/example/Foo;->value:I
  return-void
.end method
```

(b) Smali

Figure 2: Example Java file and corresponding Smali code

2.5 Instrumentation

Many black-box dynamic analysis tools modify the bytecode of the apps under analysis. This technique is called *instrumentation*. The term is somewhat overloaded, though: it is also sometimes used to describe the modification of the Android operating system itself [21]. We can again distinguish two different types of bytecode instrumentation: *static instrumentation* and *dynamic instrumentation* (not to be confused with static and dynamic *analysis*). With static instrumentation, applications are modified in one go, prior to analysis. With dynamic instrumentation on the other hand, the app is continuously modified as it runs. Dynamic instrumentation is more complex and appears to be less frequently used.

Examples of analysis tools that use static instrumentation are ICCInspect [11], AspectDroid [22], DroidFax [23], APIMonitor [24] (a system that was used in a version of DroidBox [25, 26]) and some unnamed tools [20, 27, 28]. AppTrace [29] is an example of a dynamic instrumentation tool.

The contributions of this thesis apply to only one of these two categories: the one of static instrumentation. From this point on, we will refer tools that use some form of static Dalvik bytecode instrumentation as simply *instrumentation tools*.

Some instrumentation tools analyze instrumented versions of applications in an isolated environment using automated tests. For others, applications are actually installed on user devices with instrumentation changes left in, in order to gather information while the app is being used by the end user. In the latter case, the performance overhead caused by the instrumentation is of course especially important. Our contributions apply to both of these approaches.

Instrumentation tools have a variety of purposes. DroidFax [23] and the tool developed by Somarriba et al. [27] monitor and visualize the runtime behavior of apps, the latter focusing on malware detection. ICCInspect [11] provides statistics and visualizations for the runtime usage of Android's ICC system. The tool from Hu et al. [28] analyzes the energy consumption of methods and API calls, helping developers with the optimization of their apps.

Another common use for instrumentation is *taint tracking*. Some calls to the Android API provide sensitive information about the owner of the phone, such as their contact list or location. These calls are *taint sources*; the registers that receive the data are *tainted* with information. Whenever a register receives information based on a tainted register, that register is tainted as well. API calls that send information to the outside world, for example via internet or SMS, are called *taint sinks*. Clearly, we do not want apps to send sensitive information to the outside world without the consent of the user. The Android permission system is however not always sufficient to prevent this: benign apps may very well require permissions for both information access and transmission, even if they do not intend to transmit sensitive information. Taint tracking tools detect whether any sensitive information reaches a taint sink at runtime by keeping track of which registers are tainted. AspectDroid [22], DroidBox [25] and the tool developed by Will [20] are examples of taint tracking tools.

There are also a number of frameworks for the development of instrumentation tools. Examples are Apkil [24], I-ARM-Droid [30] and InsDal [31]. An interesting system that also somewhat fits in this category is Repackman [32], which can repackage apps with arbitrary payloads in order to evaluate other tools that detect such repackaging.

Note that although we have thus far only spoken of *analysis* tools, some instrumentation tools do take it a step further: they instrument apps in order to improve them, usually focusing on security and privacy. If instrumentation is used for this purpose, it is often called *bytecode rewriting* or *app hardening*. One such tool is introduced by [33], describing the use cases of advertisement removal and the injection of a more fine-grained permission system. The system from [34] also involves bytecode instrumentation to make the Android permission system more fine-grained. Aurasium [35] is yet another example. An overview of techniques of this family is given by [36].

Finally, an interesting group of instrumentation tools is formed by tools that measure black-box code coverage, i.e. how much of the bytecode of the application under test is actually executed during analysis. As we mentioned before, the primary weakness of dynamic analysis is that it can only detect behaviors that actually occur. Knowledge of code coverage is therefore quite valuable for dynamic analysis tools, and is also useful for their evaluation. The metric is also valuable for tools that automatically generate tests, which are often used by dynamic analysis

tools. For more information about automatic testing, we refer to the systematic review by Kong et al. [37]. Examples of instrumentation tools that measure code coverage are Ella [38], CovDroid [39], the tools described in [40] and [41], and ACVTool [5]. ACVTool appears to be the most mature tool of this group, and [5] describes many more tools, alternative approaches and uses for code coverage measurements.

Although bytecode instrumentation is used by a variety of tools, it does have some significant limitations. Instrumented apps must be repackaged, and malicious apps could detect this repackaging and then not execute any malicious code (again capitalizing on the general weakness of dynamic analysis). Apps can, for example, verify their own signature: changing the bytecode of an application invalidates its signature, so instrumentations tools must re-sign them before installation. Another limitation is that instrumentation may sometimes break the application under test: Pilgun reports that instrumentation success rates (the fraction of apps that remains functional after instrumentation) of older code coverage tools lie between 36% and 65% [5]. Pilgun's own tool ACVTool has a much higher success rate, but it cannot successfully instrument every app either.

As an alternative to bytecode instrumentation, many tools (e.g. [42, 43, 44, 45, 46, 47, 48]) instead change or substitute some component of the Android operating system itself, like the Android Runtime or the API framework. Usually, these tools use an emulator to run the modified operating system. A notable disadvantage of these techniques compared to bytecode instrumentation is that the tools need to be updated as the Android OS changes. The bytecode specification is sometimes changed in updates as well, but these changes are usually fairly small.

2.6 Register management

A challenge that nearly all instrumentation tools face is the management of registers. Because Dalvik bytecode is register-based, almost any meaningful addition to it will require a register. Instrumentation code could use the existing local registers if the method already has enough of them, but unless code is added only at the beginning or at the end of the method, doing so without disturbing the original code is very difficult, and not always possible. In most cases, additional registers have to be allocated.

In principe, this can be done by simply increasing the number after `.locals` or `.registers`. However, this has a problematic side-effect: the parameter registers are always the last ones, so increasing the number of local registers will shift them upwards. The `p`-registers will still point to the correct registers after the change, so the problem does not lie there. Rather, it lies in the fact that some Dalvik instructions cannot handle the full range of registers.

A method can have a total of 65 536 registers, which are therefore indexed with 16 bits, but some instructions have register index fields of only 4 bits (16 registers) or 8 bits (256) long. According to Google [19], this was done because it is common for methods to need more than 8 registers, but uncommon for them to need more than 16.

If the parameter registers are shifted upwards because of the addition of local registers, they could be shifted from the 4-bit to the 8-bit range, or from the 8-bit to the 16-bit one. It is possible that the instructions that use them cannot handle the larger range, causing the program to become malformed. Instrumentations tools have to take this problem into consideration [5, 20, 40].

An alternative solution to the register management issue is to add entirely new functions containing the code to insert, and to add calls to these functions to existing methods instead of changing them directly [40]. This can however quickly cause the program to exceed the maximum number of methods [5, 40] (which is also 65 536 [19]). Indeed, [40], which uses this technique, reports a repackaging success rate of only 36% because of this issue.

2.7 ACVTool

Our work involves the instrumentation tool ACVTool made by Pilgun [5] in particular. As mentioned, ACVTool is a tool that measures black-box code coverage. ACVTool sets itself apart from older code coverage tools with its high instrumentation success rate (96.9%) and its ability to measure coverage at the instruction level (most other tools only measure at the method level). Pilgun also introduced the concept of *dynamic binary shrinking* with his related tool ACVCut [5]. This is a technique where code that has not been executed during testing is removed from the application. This aggressive approach removes the primary weakness of dynamic analysis entirely.

ACVTool instruments code using the Smali representation. It appends tracking code after almost every instruction in a method. Figure 3 on the next page shows how ACVTool instruments the `baz` method from our earlier example in Section 2.4. The newly inserted lines are highlighted.

ACVTool also adds the `tool/acv/AcvReporter` class to the code, which contains a Boolean array for every existing class of the application. These arrays have a value for every method and every instruction, which is set to `true` if that method or class has been covered (i.e. reached). In Figure 3b, line 5 loads the Boolean array for `Foo`, lines 6 to 8 mark the method as covered, and the other inserted lines mark each individual instruction as covered. See [5] for the specific details of the approach. ACVTool requires three additional registers per method, which is why the number of local registers of `baz` has been increased from one to four.

ACVTool solves the register management problem in a very robust way: it adds instructions at the beginning of the method that copy the values from the parameter registers back to their original positions, and replaces each occurrence of a `p`-register in the method with the `v`-register that corresponds with the original position of the `p`-register. This ensures that the registers used by the original instructions remain the same, and frees up the last three registers for ACVTool to use. This approach was described in an earlier work from Will [20] as well.

In the example from Figure 3, the `p0` and `p1` registers used to correspond to `v1` and `v2` respectively, as there was only one local register. After increasing the number of local registers to four, they correspond to `v4` and `v5`. Lines 3 and 4 of Figure 3b copy the values from `p0`/`v4` and `p1`/`v5` back to `v1` and `v2`, and each occurrence of `p0` or `p1` in the method has been replaced with `v1` or `v2` (for example in line 9).

```
.method private baz(I)V
  .locals 1
  iget v0, p0, Lcom/example/Foo;->value:I
  add-int/2addr v0, p1
  iput v0, p0, Lcom/example/Foo;->value:I
  return-void
.end method
```

(a) Original

```
.method private baz(I)V
  .locals 4
  move-object/16 v1, p0
  move/16 v2, p1
  sget-object v3, Ltool/acv/AcvReporter;->LcomexampleFoo583:[Z
  const/16 v4, 0x1
  const/16 v5, 0xe
  aput-boolean v4, v3, v5
  iget v0, v1, Lcom/example/Foo;->value:I
  goto/32 :goto_hack_2
  :goto_hack_back_2
  add-int/2addr v0, v2
  goto/32 :goto_hack_1
  :goto_hack_back_1
  iput v0, v1, Lcom/example/Foo;->value:I
  goto/32 :goto_hack_0
  :goto_hack_back_0
  return-void
  :goto_hack_0
  const/16 v5, 0xb
  aput-boolean v4, v3, v5
  goto/32 :goto_hack_back_0
  :goto_hack_1
  const/16 v5, 0xc
  aput-boolean v4, v3, v5
  goto/32 :goto_hack_back_1
  :goto_hack_2
  const/16 v5, 0xd
  aput-boolean v4, v3, v5
  goto/32 :goto_hack_back_2
.end method
```

(b) Instrumented

Figure 3: An example of ACVTool's instrumentation

After running the instrumented app, ACVTool can generate a report of which instructions have been covered by using the information gathered by the injected `ACVReporter` class.

2.8 Composition of instrumentation tools

Given the variety of instrumentation tools that are available, it may be useful to work with multiple of them at the same time. Example use cases are composing multiple analysis tools to scan for different kinds of behavior at the same time, combining a tool that looks for malicious behavior with a code coverage tool in order to gain information on how trustworthy the results are, and composing multiple app hardening tools to gain the benefits of each of them. To our knowledge, not much research has been done in this area.

For analysis tools, a simple approach to composition would be to repeat the same input for differently instrumented versions of an app. This is however not always possible: even with the same input, apps don't always behave in the same way, since they may use some form of external input like the internet, their behavior may depend on the current time, or they may just contain random elements. The time overhead of such a scheme might also be quite large. And of course, it does not apply at all in the case of app hardening.

Instrumenting an application with multiple tools may however cause problems, as instrumentation tools generally assume that no changes have been applied to the application before, and no changes will be applied after. Applying instrumentation tools one after the other will cause the later tools to instrument the code added by the earlier ones, which may result in undesired behavior: we do not want to measure the coverage of the code inserted by other tools, or to analyze such inserted code for malicious behavior. It may even lead to a combinatorial explosion of added code, significantly increasing the overhead of the instrumentation. Multiply instrumented apps may also fail to run altogether, since we already know that success rate of individual tools can be quite low.

The composition of instrumentation tools is the subject we set out to investigate in this work.

3 Instrumentation blueprints

In order to better facilitate the composition of instrumentation tools, we will now introduce our main contribution: the concept of *instrumentation blueprints*. We will first explain what blueprints are, how they address the problem of composition and why we chose this approach, and then we will describe how we designed them. Finally, we will explain their syntax in full detail.

3.1 Approach

As stated in Section 2.8, the main disadvantage of using instrumentation tools one after the other is that they will instrument each other's changes, possibly leading to problems. In order to avoid these problems, we essentially need to instrument an application with both tools *at the same time*. This is exactly what we aim to make possible by using instrumentation blueprints.

An instrumentation blueprint is a file that contains all the changes that an instrumentation tool wants to apply to the bytecode. It is essentially a kind of `diff`, but a bit richer. Our proposed composition architecture works as follows: instead of instrumenting an application directly, tools output a blueprint file with the changes they want to apply. A separate applicator program can then receive multiple of such blueprints and apply all of them at once. Because the applicator has knowledge of all the changes that need to be made, it is able to avoid certain problems that would otherwise arise, or to at least warn the user in the case that composition is not possible.

Figure 4 on the following page shows a diagram of how instrumentation tool composition works with and without instrumentation blueprints. Without blueprints, the tools are used one after the other. The use of the second tool may break the changes of the first tool, or lead to the instrumentation of the first tool's instrumentation code. With blueprints, each tool first outputs a blueprint individually, and the applicator then applies these blueprints at the same time. This allows it to avoid the issues inherent to the composition.

3.2 Motivation

Our approach has the disadvantage that it requires internal changes to existing instrumentation tools, in order to make them output blueprints. It is however highly generic: tools that generate blueprints will be more composable with any future tool that does so as well. The landscape of instrumentation tools is quite varied, and it changes rapidly: new tools appear and old ones deprecate in quick succession. A generic way to compose these tools may therefore be of great value.

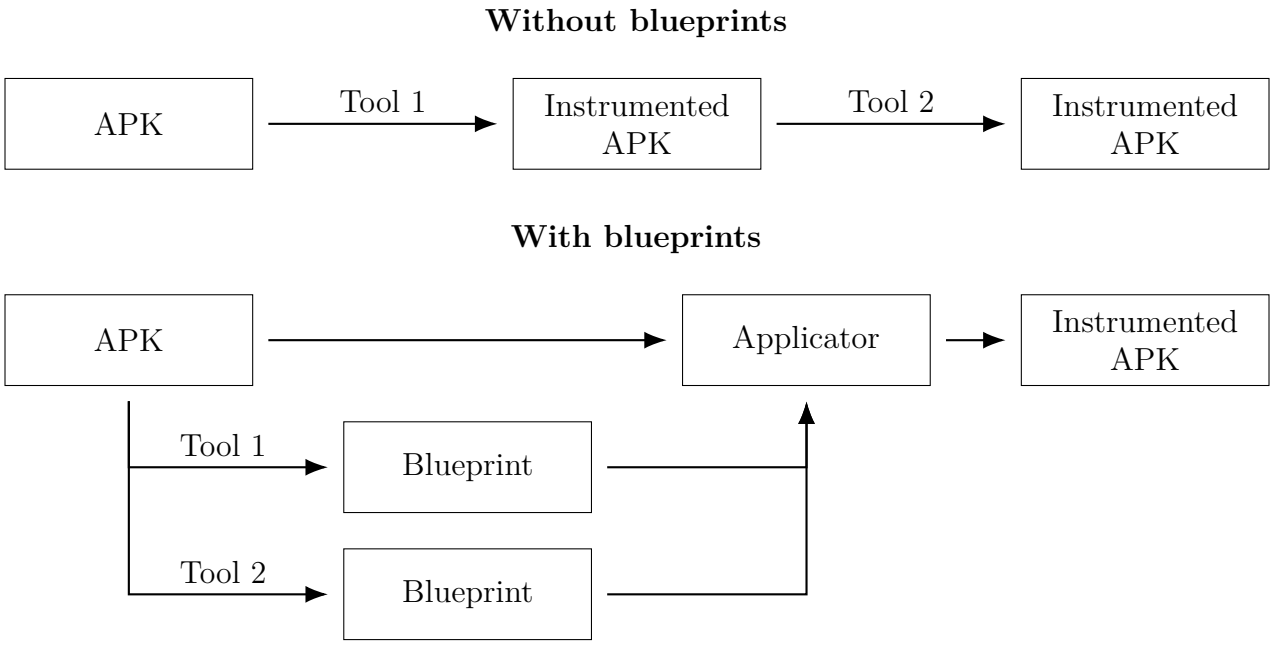


Figure 4: Process flow for instrumentation composition with and without blueprints

Blueprints may also simplify the creation of instrumentation tools. Although they are not intended as a fully-fledged instrumentation framework, they could still be a helpful abstraction on top of direct code instrumentation (for reasons that we are yet to explain). Instrumentation frameworks could be implemented on top of this intermediate abstraction level, which would have the additional benefit of automatic blueprint generation for any tool implemented with such a framework. Furthermore, blueprints may open up possibilities to meta-analyze the instrumentation methodologies of instrumentation tools, since they represent the changes they apply in a systematic way.

Initially, the direction of this thesis was more practical: studying issues that might arise from specific combinations of instrumentation tools. An additional reason for the switch to the more generic approach of instrumentation blueprints was the lack of availability of a large amount of tools. Although many tools are described in literature (as shown in Section 2.5), a great deal of them are simply not publicized. Furthermore, those that are available often do not run without hiccups. Publicized tools are often poorly documented, and time limitations did not allow us to figure out how to properly install and use enough of them to execute a thorough practical analysis of issues caused by composition. Instrumentation blueprints are both a generalization and specialization of the topic of instrumentation composition: they can be used with any instrumentation tool, but they can also be implemented and tested using only a single one of them.

3.3 Blueprint design

Our main goal when designing a prototype syntax for instrumentation blueprints was to make them highly expressive in order to support as many instrumentation tools as possible, while at the same time giving them enough structure to actually help with composition.

Blueprints represent code using the Smali representation, because it makes the code human-readable and easier to work with while still remaining very close to the original bytecode. It may be harder to implement blueprint output for instrumentation tools that are based on a more abstract representation like Jimple, but since all representations must eventually be converted back to bytecode, it should still be possible. Converting changes at a lower-level representation to a higher-level would certainly be more difficult.

The blueprint syntax is line-based: the smallest unit whose change can be represented is a single line of Smali code. Lines can can be changed in three distinct ways: code can be *prepended* to them, *appended* to them, or they can be entirely *replaced*. In principle, all code changes can be represented using replacements (or by using additions and deletions like `diff`), but including the intention behind the change is what allows us to compose multiple blueprints.

Distinguishing prepend-additions from append-additions may also seem superfluous, as appending to line `n` is equivalent to prepending to line `n+1`. However, when multiple blueprints are combined, the difference can actually be meaningful. For example, a tool could append a (conditional) jump instruction after line `n` that may cause code prepended to line `n+1` to not be reached. Again, we aim to capture the intention behind the code changes, and separating *prepend* from *append* yields a bit more expressivity in that regard.

Currently, the only requirement for two blueprints to be composable is that they do not include a replacement for the same line. Instrumentation tools should therefore refrain from using the replacement option whenever possible. We believe that most instrumentation tools do not need to replace lines, since they usually aim to analyze the code that already exists in a transparent manner (i.e. without changing its behavior). We expect that replace-conflicts are only unavoidable when tools are inherently not composable, for example when two app hardening tools try to modify the same part of a program, but we did not investigate this thoroughly.

There is however one important exception to this: tools may need to replace lines of code whose behavior they don't intend to change for the purpose of register management. For example, as we explained in Section 2.7, ACVTool needs to change every line that contains a parameter register. For this reason, we designed the blueprint syntax to abstract register management away.

Blueprints consist of a series of *method entries*, each containing the line changes for a single method. Every method entry specifies how many additional registers the instrumentation code needs. The included smali code can then refer to these additional *instrumentation registers* using the names `i0`, `i1`, `i2` and so on (the `i` stands for instrumentation). Of course, the normal `v`- and `p`-registers can still be used as well. The applicator program will then ensure that the registers are managed correctly (the way in which it does this will be described in Section 4.2).

3.4 The syntax

All instrumentation changes to an APK file are condensed into a single blueprint file. As we already touched upon, blueprint consist of a list of *method entries*. These method entries consist of a header, followed by a list of *line entries*. Line entries have a header as well, and optionally Smali code contents.

The format of these entries is shown in Figure 5. The `<method>` field specifies the fully qualified descriptor of the method, and the `<register-count>` field specifies the required number of instrumentation registers. Lines are identified by their line number relative to the method (starting at zero), which is placed in the `<line-number>` field. The `<operation>` field contains a character that identifies type of line operation. The options are shown in Table 2. Finally, the `<content>` field may consist of any amount of Smali instructions, optionally using `i`-registers. Multiple method entries for the same method, or multiple line entries for the same line and operation, are permitted.

```
@@<method>:<register-count>
<line entry>
<line entry>
...
```

(a) Method entry

```
@<line-number>:<operation>
<content>
```

(b) Line entry

Figure 5: Formats of method and line entries

Character	Operation	Description
`a`	Append	Add `<content>` after the line
`p`	Prepend	Add `<content>` before the line
`r`	Replace	Replace the line with `<content>`

Table 2: Possible line entry operations

Both method and line entry headers can appear directly after a line of Smali code, so we have to be able to distinguish these headers from Smali. This is achieved by beginning both headers with an `@`-character, since beginning a line with one is not legal in Smali. Its "at"-meaning also fits rather well. Method entry headers have an additional `@` to distinguish them from line entry headers.

Because we identify lines using their line number, we need to be very precise about which lines are counted. Generally, the fewer lines are counted, the easier the implementation of blueprint output for instrumentation tools becomes, but changes to lines that are not counted cannot be represented in a blueprint. We decided to count every line, except for the following ones:

- Empty lines.
- The line containing `.locals` or `.registers`.
- Lines containing debug information.

The lines that we consider to be debug lines are those containing a `.line`, `.local` or `.prologue` directive. We do not count these lines because they are optional, they do not alter the state of the program, and we cannot think of any reason to instrument them: they are equivalent to empty lines. Any change to a debug line can instead be represented as a prepend entry for the line that comes after it.

The syntax is still a prototype: there are multiple code modifications that it currently cannot represent. We will discuss these shortcomings in Section 6. A concrete example of the blueprint syntax will be given in Section 4.1.

4 Implementation

We have implemented our blueprint system from two directions: we extended ACVTool to generate blueprints, and we created a program that can apply blueprints to Smali files. In this section, we describe how we went about each of these directions.

4.1 Generation of instrumentation blueprints for ACVTool

We have extended ACVTool to generate a blueprint as a side-effect of instrumentation. Because ACVTool uses Smali and instruments almost every line, it is a good test of both the expressivity of our syntax and the correctness of our applicator (Section 4.2).

ACVTool is written in Python and its source code is publicly available [49]. Although the code contains very little documentation, it is fairly self-explanatory. The detailed explanation of the instrumentation process in [5] was also very helpful for understanding the purpose of certain parts of the code. The tool uses a modified version of Apkil, a bytecode instrumentation library that was originally created for APIMonitor [24]. Apkil discards lines containing debug information at an early stage, which partly influenced our decision to not count those lines for the blueprint syntax.

We identified all locations in ACVTool's code where Smali was inserted into the application and added blueprint generation code for each of them. ACVTool already created an auxillary `.pickle` file, which it used to generate a report from the analysis results. We made ACVTool additionally create a blueprint file at the same location.

Figure 6 on the next page shows the blueprint segment corresponding to the ACVTool-instrumentation example from Figure 3. The lines that are highlighted in Figure 6b which also appear in Figure 6c are highlighted there as well.

The blueprint begins with a header specifying the method `baz(I)V` from `com/example/Foo`. ACVTool needs three instrumentation registers per method, so the header ends with `:3`. Below that, the blueprint contains five line entries: `@0:p`, `@0:a`, `@1:a`, `@2:a` and `@3:a`. The first entry contains the prepended lines that load in the coverage array and mark the method as covered. The other entries contain the appended lines that mark each of the original instructions as covered. Note that the blueprint uses the `i0`, `i1` and `i2` registers where the instrumented code uses `v3`, `v4` and `v5`. The first two instructions added by ACVTool are omitted, since they only served to copy the values of the parameters to their original positions, in order to free up the `v3`, `v4` and `v5` registers (as explained in Section 2.7). Since register management has been abstracted away by the `i`-register system, these two instructions should not be included in the blueprint.

```
.method private baz(I)V
  .locals 1
  iget v0, p0, Lcom/example/Foo;->value:I
  add-int/2addr v0, p1
  iput v0, p0, Lcom/example/Foo;->value:I
  return-void
.end method
```

(a) Original bytecode

```
.method private baz(I)V
  .locals 4
  move-object/16 v1, p0
  move/16 v2, p1
  sget-object v3, Ltool/acv/AcvReporter;->
    ↪ LcomexampleFoo583:[Z
  const/16 v4, 0x1
  const/16 v5, 0xe
  aput-boolean v4, v3, v5
  iget v0, v1, Lcom/example/Foo;->value:I
  goto/32 :goto_hack_2
  :goto_hack_back_2
  add-int/2addr v0, v2
  goto/32 :goto_hack_1
  :goto_hack_back_1
  iput v0, v1, Lcom/example/Foo;->value:I
  goto/32 :goto_hack_0
  :goto_hack_back_0
  return-void
  :goto_hack_0
  const/16 v5, 0xb
  aput-boolean v4, v3, v5
  goto/32 :goto_hack_back_0
  :goto_hack_1
  const/16 v5, 0xc
  aput-boolean v4, v3, v5
  goto/32 :goto_hack_back_1
  :goto_hack_2
  const/16 v5, 0xd
  aput-boolean v4, v3, v5
  goto/32 :goto_hack_back_2
.end method
```

(b) Instrumented bytecode

```
@@Lcom/example/Foo;->baz(I)V:3
@0:p
sget-object i0, Ltool/acv/AcvReporter;->
    ↪ LcomexampleFoo583:[Z
const/16 i1, 0x1
const/16 i2, 0xe
aput-boolean i1, i0, i2
@0:a
goto/32 :goto_hack_2
:goto_hack_back_2
@1:a
goto/32 :goto_hack_1
:goto_hack_back_1
@2:a
goto/32 :goto_hack_0
:goto_hack_back_0
@3:a
:goto_hack_0
const/16 i2, 0xb
aput-boolean i1, i0, i2
goto/32 :goto_hack_back_0
:goto_hack_1
const/16 i2, 0xc
aput-boolean i1, i0, i2
goto/32 :goto_hack_back_1
:goto_hack_2
const/16 i2, 0xd
aput-boolean i1, i0, i2
goto/32 :goto_hack_back_2
```

(c) Blueprint segment

Figure 6: An example blueprint segment for ACVTool

4.2 Blueprint applicator

Besides designing a prototype blueprint syntax and extending ACVTool to generate blueprints, we also created the prototype blueprint applicator program `applybp`. We wrote the program in C++. It has two functions: `apply` and `merge`. The primary function `apply` is capable of applying any amount of blueprints to a specified set of Smali files. The additional function `merge` merges multiple blueprints into a single one and outputs the result. This allows us to examine the result of combining two blueprints without actually applying them.

For either function, before looking at any Smali file, `applybp` first parses all specified blueprint files and merges them into a single data structure. We do this primarily for register management purposes and to detect incompatible blueprints early, but it has performance advantages as well. If the original program consists of n lines, and the blueprints to apply have a total sum of m line entries, the simple approach of looking up all line entries that affect a smali line for every line would result in a time complexity of $O(n \cdot m)$. By first merging all blueprints into a single data structure, we can improve on this.

The blueprint syntax has a natural "method entry → Smali line number → line operation" tree structure. The blueprint data structure stores this tree using lookup maps (`std::map`). Creating the structure therefore has a time complexity of $O(m \log(m))$: inserting an element into the tree has a complexity of $O(\log(m))$, and there are m lines to insert. After parsing all the blueprint files, applying them to the smali code has a time complexity of $O(n \log(m))$: looking up a line entry in the data structure is logarithmic. The total complexity therefore becomes $O(m \log(m) + n \log(m))$. If we assume that m grows about as fast as n, which seems realistic (more lines means more instrumented lines), then $O(m \log(m) + n \log(m)) = O(n \log(n))$, which is better than $O(n \cdot m) = O(n^2)$.

If multiple method entries for the same method are encountered, they are merged together. When method entry B is merged into method entry A, the instrumentation register count of A is set to the sum of the counts of A and B. Every line entry from B is added to A, but all instrumentation register indices are increased with the original instrumentation register count of A. For example, if A used three instrumentation registers (`i0`, `i1` and `i2`) and B used two (`i0` and `i1`), the combined method entry uses five, and all line entries that came from B refer to `i3` and `i4` instead of `i0` and `i1`. This ensures that the added lines from each of the method entries do not affect each other.

If multiple line entries for the same line and operation type (*append*/*prepend*/*replace*) are encountered, one of two things happens: If the operation is *append* or *prepend*, the Smali contents are simply concatenated. However, like we stated in Section 3.3, if there are two replacements for the same line, the blueprints are considered non-composable, and `applybp` aborts with an error message.

Note that the blueprint syntax does not prohibit multiple method entries for the same method or multiple line entries for the same line, so merges (and even replace-conflicts) can occur within a single blueprint. In fact, concatenating multiple blueprint files and then passing them to `applybp` as one large file is equivalent to passing them separately. Using `applybp`'s `merge` function with only a single blueprint as input will squash all duplicate entries.

If the `merge` function was chosen, `applybp` prints the result from the merge and exits. Figure 7 on the following page shows an example result of merging two blueprints. Both blueprints have a method entry for `Lcom/example/Foo;->bar(I)V`. Blueprint 1's version uses three instrumentation registers and blueprint 2's version uses two. In the merged blueprint, this method entry therefore uses $3 + 2 = 5$ of them. The `@0:a` line entry from blueprint 2 is is added to the `@0:p` and `@0:r` line entries from blueprint 1 without problems. Both `Lcom/example/Foo;->bar(I)V` method entries have a `@1:a` line entry, so the merged blueprint

contains the contents of both. Note how the indices of all instrumentation registers used by the line entry contents that came from blueprint 2's `Lcom/example/Foo;->bar(I)V` method entry have been incremented by three. A method entry for `baz` only appears in blueprint 2, so it is included in the merged blueprint without any modifications.

```
@@Lcom/example/Foo;->bar(I)V:3
@0:p
add-int i0, i1, i2
@0:r
sub-int i0, i1, i2
@1:a
mul-int i0, i1, i2
```

(a) Blueprint 1

```
@@Lcom/example/Foo;->bar(I)V:2
@0:a
div-int i0, i1, v0
@1:a
rem-int i0, i1, v0
@@Lcom/example/Foo;->baz(I)V:1
@0:a
neg-int i0, v0
```

(b) Blueprint 2

```
@@Lcom/example/Foo;->bar(I)V:5
@0:p
add-int i0, i1, i2
@0:r
sub-int i0, i1, i2
@0:a
div-int i3, i4, v0
@1:a
mul-int i0, i1, i2
rem-int i3, i4, v0
@@Lcom/example/Foo;->baz(I)V:1
@0:a
neg-int i0, v0
```

(c) Merged

Figure 7: The result of merging two blueprints

If the `apply` function was chosen, `applybp` will proceed with applying the merged blueprint to the specified Smali targets. Targets can be either files or directories: in the case of directory, `applybp` applies the blueprints to all files in the directory recursively.

To manage registers, `applybp` uses the same method as ACVTool [5] and the tool described by Will [20], because Pilgun has shown that this method is very robust [5]. We increment the number of local registers by the amount of instrumentation registers, then copy the values of the parameters to the `v`-registers corresponding to their original positions, and then replace all `p`- and `i`-register references with their `v`-equivalents.

Figure 8 illustrates the register management process for an example method with one original local register, two parameter registers and three instrumentation registers. Initially, the method has three registers in total, and `p0` and `p1` are aliases of `v1` and `v2`. After incrementing the local register count with three, there are now five registers in total, and the parameter registers point to `v4` and `v5`. The values of the parameter registers are then copied back to `v1` and `v2`, leaving `v3`, `v4` and `v5` available as instrumentation registers.

```
v0
v1 = p0
v2 = p1
```

(a) Initial layout

```
v0
v1
v2
v3
v4 = p0
v5 = p1
```

(b) Additional local registers

```
v0
v1 = p0
v2 = p1
v3 = i0
v4 = i1
v5 = i2
```

(c) Parameters moved

Figure 8: Register management process

When applying the merged blueprint, `applybp` will read the Smali files line by line, generally copying them directly to its output. When it encounters a method, it will look up if the blueprint contains an entry for it, and if it does, it will apply its line entries. The number in the `.locals`/`.registers` line is incremented as specified by the method entry, and move instructions are added to move the parameters to `v`-registers. Dalvik contains a few different move instructions; the specific one to use depends on the type of the parameter.

The application of line entries is fairly straightforward: prepend contents are added before the line, append contents are added after, and if there is a replace entry, the line is replaced with its contents. For every line of Smali that is written in an instrumented method — whether it comes from the original code or from the blueprint — all `p`- and `i`-registers are replaced with their `v`-equivalents (as in Figure 8c).

After applying the merged blueprint to a file, the result is written to a file with the same name in a user-specified output directory. If a directory of Smali files is specified as a target, the directory structure is mirrored in the output directory.

Our program needs to parse two languages: the blueprint language, and Smali (the blueprint language also contains a subset of the Smali language). We wrote two simple recursive descent parsers for this purpose. Our Smali parser is very limited: it only parses exactly what `applybp` needs to function, and leaves everything else as strings. An advantage of this is that the parser is fairly future-proof. For example, it does not care about specific instructions, so it will not be affected if new instructions are added to Dalvik.

We ran into quite a few issues while implementing the application part of `applybp`, mostly because the Smali syntax lacks extensive documentation. We used the Android Emulator in combination with the debug tool `logcat` [50] to discover and fix any issues we came across. Some notable examples are:

- Two of the types supported Smali, *long* and *double*, are "wide": their values occupy two registers instead of one. We had to take this into account for the code that copies `p`-registers to `v`-registers. Figure 9 on the following page shows an example register layout with one local register and three parameters, where the first and third parameters are wide types. Note that the `p1` name is skipped.
- Methods that are `abstract` or `native` (implemented in native code) are empty in Smali: they do not even contain a `.locals`/`.registers` line. Our program ignores these methods when applying blueprints.
- Instead of the `operation arg1, arg2, arg3` syntax that is used by almost all Smali instructions, method calls use lists of registers. For example: `invoke-direct {p0, v0}, <method-descriptor>` (see Figure 2b on page 6). The variant `{v0 .. v3}` is sometimes used as well.
- Before we clearly defined our policy of which lines are counted for the purpose of blueprint line entry line numbers, some "block-directives" caused the counts of `applybp` and our ACVTool blueprint output to become mismatched. An example of such a block-directive is `.packed-switch`/`.end packed-switch`, which corresponds to the packed-switch-payload as described in [19].

```
v0
v1 = p0
v2
v3 = p2
v4 = p3
v5
```

Figure 9: Example register layout with wide parameters `p0` and `p3`

Our program is still a prototype, and as such, it still has a few limitations. We will discuss these limitations in Section 6.

5 Evaluation

We tested the correctness of our ACVTool blueprint-generation extension and our blueprint application program `applybp` using a few apps from F-droid[1]. We generated blueprints for the apps, applied them to the originals using `applybp` and checked whether the resulting apps ran without problems on the Android Emulator.

Because of a number of limitations of our prototype blueprint syntax (which we will further discuss in Section 6), it is currently not actually capable of representing all of ACVTool's app changes. Two changes that cannot be represented are that ACVTool adds some entries to the manifest file, and that it injects a few extra classes (like `tool/acv/AcvReporter`). In order to evaluate the application of the changes that our syntax does support (changes to method contents), we used a procedure that circumvents these limitations:

1. We instrument the original app with ACVTool, which generates a blueprint as a side-effect.
2. We unpack the original app with Apktool.
3. We apply the blueprint to the Smali files of the original app with `applybp`.
4. We unpack the ACVTool-instrumented app with Apktool.
5. We copy the Smali files emitted by `applybp` to the unpacked ACVTool-instrumented app, overwriting existing ones.
6. We repack, re-sign and install the resulting app.

The manifest file changes and the added files did therefore not pass through our system.

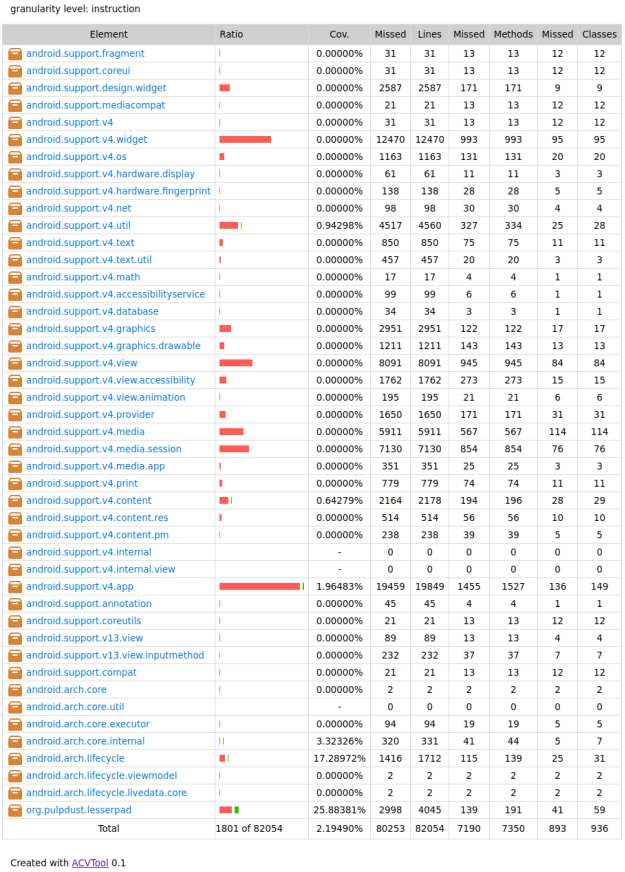
After several rounds of bugfixes, all examined F-droid apps that ran successfully with ACVTool's instrumentation also did so after being instrumented according the the above procedure.

To verify whether ACVTool's coverage-measuring code still functioned correctly when applied through `applybp`, we generated a code coverage report with both a directly instrumented version and an `applybp`-instrumented version of *Lesser Pad*, a simple note-taking app from F-droid[2]. For both versions, we installed the app on the Android Emulator, opened it, interacted with it for a few seconds, closed it, and then made ACVTool generate a coverage report using the gathered data. The experiment was rather informal: we did not use a testing framework to repeat the exact same inputs for each version.

Figure 10 on the following page shows screenshots of the top levels of both generated web reports. As can be seen at a glance, both reports are virtually the same.

Although we did not perform extensive systematic experiments, these results suggest that our ACVTool blueprint-generation and `applybp` blueprint application implementations are indeed correct.

[1] `https://www.f-droid.org/`
[2] `https://f-droid.org/en/packages/org.pulpdust.lesserpad/`

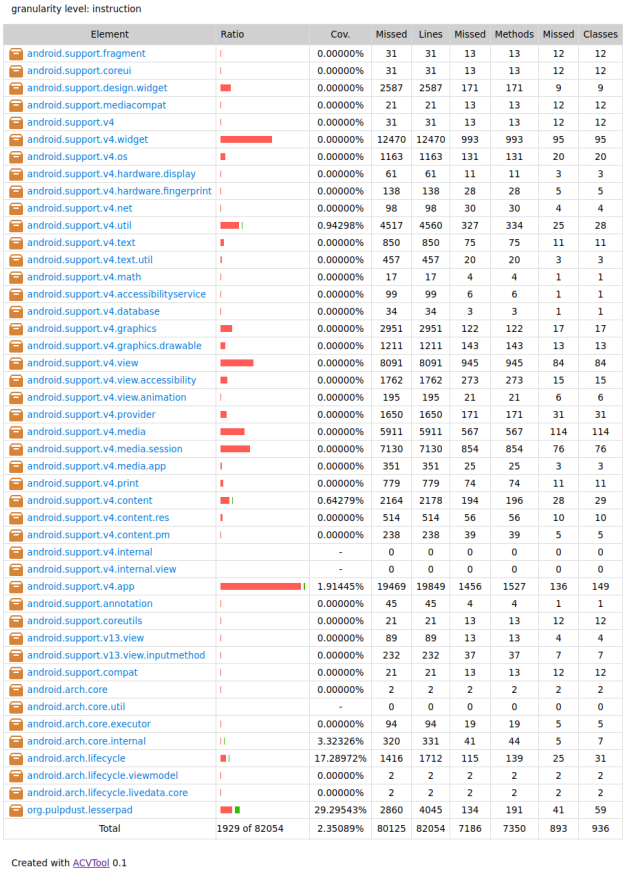


granularity level: instruction

Element	Ratio	Cov.	Missed	Lines	Missed	Methods	Missed	Classes
android.support.fragment		0.00000%	31	31	13	13	12	12
android.support.coreui		0.00000%	31	31	13	13	12	12
android.support.design.widget		0.00000%	2587	2587	171	171	9	9
android.support.mediacompat		0.00000%	21	21	13	13	12	12
android.support.v4		0.00000%	31	31	13	13	12	12
android.support.v4.widget		0.00000%	12470	12470	993	993	95	95
android.support.v4.os		0.00000%	1163	1163	131	131	20	20
android.support.v4.hardware.display		0.00000%	61	61	11	11	3	3
android.support.v4.hardware.fingerprint		0.00000%	138	138	28	28	5	5
android.support.v4.net		0.00000%	98	98	30	30	4	4
android.support.v4.util		0.94298%	4517	4560	327	334	25	28
android.support.v4.text		0.00000%	850	850	75	75	11	11
android.support.v4.text.util		0.00000%	457	457	20	20	3	3
android.support.v4.math		0.00000%	17	17	4	4	1	1
android.support.v4.accessibilityservice		0.00000%	99	99	6	6	1	1
android.support.v4.database		0.00000%	34	34	3	3	1	1
android.support.v4.graphics		0.00000%	2951	2951	122	122	17	17
android.support.v4.graphics.drawable		0.00000%	1211	1211	143	143	13	13
android.support.v4.view		0.00000%	8091	8091	945	945	84	84
android.support.v4.view.accessibility		0.00000%	1762	1762	273	273	15	15
android.support.v4.view.animation		0.00000%	195	195	21	21	6	6
android.support.v4.provider		0.00000%	1650	1650	171	171	31	31
android.support.v4.media		0.00000%	5911	5911	567	567	114	114
android.support.v4.media.session		0.00000%	7130	7130	854	854	76	76
android.support.v4.media.app		0.00000%	351	351	25	25	3	3
android.support.v4.print		0.00000%	779	779	74	74	11	11
android.support.v4.content		0.64279%	2164	2178	194	196	28	29
android.support.v4.content.res		0.00000%	514	514	56	56	10	10
android.support.v4.content.pm		0.00000%	238	238	39	39	5	5
android.support.v4.internal		-	0	0	0	0	0	0
android.support.v4.internal.view		-	0	0	0	0	0	0
android.support.v4.app		1.96483%	19459	19849	1455	1527	136	149
android.support.annotation		0.00000%	45	45	4	4	1	1
android.support.coreutils		0.00000%	21	21	13	13	12	12
android.support.v13.view		0.00000%	89	89	13	13	4	4
android.support.v13.view.inputmethod		0.00000%	232	232	37	37	7	7
android.support.compat		0.00000%	21	21	13	13	12	12
android.arch.core		0.00000%	2	2	2	2	2	2
android.arch.core.util		-	0	0	0	0	0	0
android.arch.core.executor		0.00000%	94	94	19	19	5	5
android.arch.core.internal		3.32326%	320	331	41	44	5	7
android.arch.lifecycle		17.28972%	1416	1712	115	139	25	31
android.arch.lifecycle.viewmodel		0.00000%	2	2	2	2	2	2
android.arch.lifecycle.livedata.core		0.00000%	2	2	2	2	2	2
org.pulpdust.lesserpad		25.88381%	2998	4045	139	191	41	59
Total	1801 of 82054	2.19490%	80253	82054	7190	7350	893	936

Created with ACVTool 0.1

(a) Direct instrumentation



granularity level: instruction

Element	Ratio	Cov.	Missed	Lines	Missed	Methods	Missed	Classes
android.support.fragment		0.00000%	31	31	13	13	12	12
android.support.coreui		0.00000%	31	31	13	13	12	12
android.support.design.widget		0.00000%	2587	2587	171	171	9	9
android.support.mediacompat		0.00000%	21	21	13	13	12	12
android.support.v4		0.00000%	31	31	13	13	12	12
android.support.v4.widget		0.00000%	12470	12470	993	993	95	95
android.support.v4.os		0.00000%	1163	1163	131	131	20	20
android.support.v4.hardware.display		0.00000%	61	61	11	11	3	3
android.support.v4.hardware.fingerprint		0.00000%	138	138	28	28	5	5
android.support.v4.net		0.00000%	98	98	30	30	4	4
android.support.v4.util		0.94298%	4517	4560	327	334	25	28
android.support.v4.text		0.00000%	850	850	75	75	11	11
android.support.v4.text.util		0.00000%	457	457	20	20	3	3
android.support.v4.math		0.00000%	17	17	4	4	1	1
android.support.v4.accessibilityservice		0.00000%	99	99	6	6	1	1
android.support.v4.database		0.00000%	34	34	3	3	1	1
android.support.v4.graphics		0.00000%	2951	2951	122	122	17	17
android.support.v4.graphics.drawable		0.00000%	1211	1211	143	143	13	13
android.support.v4.view		0.00000%	8091	8091	945	945	84	84
android.support.v4.view.accessibility		0.00000%	1762	1762	273	273	15	15
android.support.v4.view.animation		0.00000%	195	195	21	21	6	6
android.support.v4.provider		0.00000%	1650	1650	171	171	31	31
android.support.v4.media		0.00000%	5911	5911	567	567	114	114
android.support.v4.media.session		0.00000%	7130	7130	854	854	76	76
android.support.v4.media.app		0.00000%	351	351	25	25	3	3
android.support.v4.print		0.00000%	779	779	74	74	11	11
android.support.v4.content		0.64279%	2164	2178	194	196	28	29
android.support.v4.content.res		0.00000%	514	514	56	56	10	10
android.support.v4.content.pm		0.00000%	238	238	39	39	5	5
android.support.v4.internal		-	0	0	0	0	0	0
android.support.v4.internal.view		-	0	0	0	0	0	0
android.support.v4.app		1.91445%	19469	19849	1456	1527	136	149
android.support.annotation		0.00000%	45	45	4	4	1	1
android.support.coreutils		0.00000%	21	21	13	13	12	12
android.support.v13.view		0.00000%	89	89	13	13	4	4
android.support.v13.view.inputmethod		0.00000%	232	232	37	37	7	7
android.support.compat		0.00000%	21	21	13	13	12	12
android.arch.core		0.00000%	2	2	2	2	2	2
android.arch.core.util		-	0	0	0	0	0	0
android.arch.core.executor		0.00000%	94	94	19	19	5	5
android.arch.core.internal		3.32326%	320	331	41	44	5	7
android.arch.lifecycle		17.28972%	1416	1712	115	139	25	31
android.arch.lifecycle.viewmodel		0.00000%	2	2	2	2	2	2
android.arch.lifecycle.livedata.core		0.00000%	2	2	2	2	2	2
org.pulpdust.lesserpad		29.29543%	2860	4045	134	191	41	59
Total	1929 of 82054	2.35089%	80125	82054	7186	7350	893	936

Created with ACVTool 0.1

(b) Blueprint application

Figure 10: ACVTool coverage reports for differently instrumented versions of Lesser Pad

We must note that the performance of the blueprint parsing step of `applybp` is rather bad. We expected the difference in speed of the parsing and the application steps to be a small constant factor (see the time complexity discussion in Section 4.2). The blueprint application step is virtually instant, so we expected the parsing step to be similarly fast. However, the parsing step takes significantly longer. On our machine, it took ACVTool 8.29 seconds to instrument Lesser Pad (including the unpacking, repacking and re-signing steps) and it took our program 9.91 seconds to parse the blueprint. As the size of the instrumented application increases, the blueprint application time seems to grow faster than the instrumentation time, but we did not investigate this in detail.

The bad blueprint parsing performance may be caused by the fact that the blueprint files generated by ACVTool are extremely large: the blueprint for Lesser Pad consisted of 619 235 lines. The reason for this size is that ACVTool instruments every method, even those from additional libraries provided by Google. Only 29 384 of the 619 235 blueprint lines (about 5%) were for Lesser Pad-specific code. Perhaps the blueprint parsing performance could be improved if blueprints were split into separate files for every class. Do note that the bad parsing performance is not a huge issue, since it only affects the offline blueprint application time. There is no difference in the runtime performance of directly instrumented and `applybp`-instrumented apps.

We expect that our composition system will work for a large amount of instrumentation tools. Most tools aim to be transparent, i.e., to not affect the original functioning of the app. For those tools, we expect that every individual inserted (appended or prepended) block of code does not affect the state of the original code. It is in theory possible that this is not the case: for example, a tool could insert a block of code that modifies the value of a register which is not used by the next original line, only to restore the register's value in a block of code inserted after that line. However, it seems very unlikely to us that transparent instrumentation tools would use such manipulations. Furthermore, our instrumentation register system ensures that the internal states of the code added by each instrumentation tool are not mixed: the code added by one tool cannot affect the instrumentation registers of the code added by another tool. We therefore expect that insertions from transparent tools will not affect each other.

On the other hand, replacements may lead to problems, since our system does not allow the composition of two tools that replace the same line. We expect that most transparent instrumentation tools do not use replacements, since replacements usually change the behavior of the app. There is however one transparent use of replacement that we did come across: APIMonitor [24] and I-ARM-Droid [30] both insert code by replacing function calls with calls to a new function that contains both the code to insert and the original function call. Our system cannot compose two tools that use this pattern, even if their additions should theoretically not conflict with each other. Still, these tools do not *have* to use this technique: their changes can also be represented as insertions, and if they are, there is no problem. Using normal insertions instead of additional methods may even be more robust, since it does not risk exceeding the maximum amount of methods.

Our system may run into issues when non-transparent tools are involved. If two tools attempt to change the same parts of the code by means of replacements (or even deletions), our system cannot compose the tools. However, we expect that in such cases, it would make more sense to apply the tools one after the other rather than using our composition system, since each tool is in fact interested in the changes applied by the other. When we aim to compose two tools and one of them intends to change the app, it seems logical to apply this one first, and the transparent one afterwards. For example, to compose ACVCut (which removes parts of an app's code) and ACVTool, we can simply apply ACVCut first and ACVTool afterwards, since we are interested in code coverage information of the modified app. When we aim to compose multiple tools that change an app's behavior, we should probably apply them one after the other as well.

6 Limitations

Although we believe that our blueprint composition approach is promising, it does have a number of limitations. Our prototype blueprint syntax and applicator program still have some limitations as well.

Our blueprint system can only be used with static instrumentation, since all instrumentation changes have to be known before they are combined. It also requires internal changes to existing instrumentation tools (although a limited form of automatic blueprint generation may be possible, which we discuss in Section 7). Furthermore, since we use the Smali representation, the implementation of blueprint output will be more difficult for tools that are based on other representations like Jimple. We stated our reasons for using Smali in Section 3.3. It may be possible to integrate a translation of Jimple changes to Smali changes into our system, since Jimple is more high-level than Smali. Another limitation of our system is that it does not support instrumentation of native code. Integrating native code modifications may be possible, but would require a substantial amount of additional work.

We already touched upon the limitations of our prototype blueprint syntax in Section 5. These limitations are particularly significant. Blueprints can currently only represent changes to method contents: they cannot represent changes to classes, method descriptors, fields or any other components of Smali. They also cannot represent changes to an application's manifest file. Many instrumentation tools need to change the manifest file in order to function. Our current blueprint prototype cannot represent added or removed files either.

These syntax limitations can likely all be alleviated by extending the blueprint syntax. Special entry types could be added to represent added or removed methods, fields or classes (files), and method entry headers could be given additional fields for information such as return value and parameter modifications. The manifest file has a well-defined structure, so a more semantical syntax could be created to represent changes to it, with entries such as "add `<contents>` to `<xml element>`".

A minor limitation of our prototype applicator program is that application unpacking and repacking are currently not built in: it can only operate on Smali files or directories thereof. Users have to manually unpack, repack, re-sign and install `applybp`-instrumented apps. This shortcoming can be addressed with updates to the program. Another minor limitation is the bad blueprint parsing performance. This could perhaps be improved by further optimizing the program or by redesigning blueprints to use multiple files.

Finally, a general limitation of our work is that we did not perform extensive experiments, and that the experiments we did perform only involved a single instrumentation tool. We therefore do not have empirical evidence that shows whether our approach works for most tools, nor whether it actually improves the success rate of instrumentation composition.

7 Conclusions and Further Research

To address problems that may occur from the composition of instrumentation tools, we have proposed a two-step approach involving instrumentation blueprints and the application thereof. We have defined a prototype syntax for these blueprints, we have extended the code coverage tool ACVTool [5] to emit blueprints, and we have implemented the program `applybp` that can apply them.

We have performed a limited set of experiments that show that our approach can work in practice, although a number of limitations still apply. Our proposed blueprint system may also offer benefits for the creation of new instrumentation frameworks or the meta-analysis of instrumentation tools.

There are still many aspects that can be explored in future work. First of all, as explained in Section 6, a large amount of the limitations of our prototype implementation can be alleviated. Doing so would improve the practical usability of our system. Furthermore, we have only implemented blueprint output for a single instrumentation tool, so a larger-scale investigation of the effectiveness of our system, involving multiple instrumentation tools, is still in order.

Another interesting subject to explore is the automatic generation of blueprints, based on comparing the original app with the instrumented one. This would remove the need to change existing instrumentation tools internally, which would again enhance the usability of our system. Although automatically generated blueprints may not represent the intention behind the changes with perfect accuracy (for example, prepended and appended lines are impossible to distinguish after the fact), they could still be useful.

Finally, more research could be done towards discovering the problems that arise from the composition of instrumentation tools in practice. Our system currently only addresses various theoretical problems, like tools instrumenting lines added by other tools, but it provides a base for future enhancements. With more empirical knowledge of composition conflicts, our blueprint syntax and application program could be improved to solve more of them.

References

[1] S. O'Dea. *Number of smartphone subscriptions worldwide from 2016 to 2026*. Statista. July 7, 2021. URL: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (visited on 07/21/2021).

[2] S. O'Dea. Statista. June 29, 2021. URL: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (visited on 07/21/2021).

[3] *Mobile Operating System Market Share Worldwide*. StatCounter. June 2021. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide (visited on 07/21/2021).

[4] *Average number of new Android app releases via Google Play per month from March 2019 to February 2021*. Statista. July 6, 2021. URL: https://www.statista.com/statistics/1020956/android-app-releases-worldwide/ (visited on 07/21/2021).

[5] Aleksandr Pilgun. "Instruction Coverage for Android App Testing and Tuning". Ph.D. dissertation. University of Luxembourg, Nov. 2020.

[6] *Android Developers - Platform Architecture*. Google. May 7, 2020. URL: https://developer.android.com/guide/platform (visited on 03/05/2021).

[7] *Application Fundamentals*. Google. Feb. 23, 2021. URL: https://developer.android.com/guide/components/fundamentals (visited on 07/08/2021).

[8] Julian Schütte, Rafael Fedler, and Dennis Titze. "ConDroid: Targeted Dynamic Analysis of Android Applications". In: *2015 IEEE 29th International Conference on Advanced Information Networking and Applications (AINA)*. Vol. 1. IEEE Computer Society, Mar. 2015, pp. 571–578. DOI: 10.1109/AINA.2015.238.

[9] Rafael Fedlerand Marcel Kulicke and Julian Schütte. "Native Code Execution Control for Attack Mitigation on Android". In: *Proceedings of the ACM Conference on Computer and Communications Security*. Nov. 2013. DOI: 10.1145/2516760.2516765.

[10] Min Zheng, Mingshen Sun, and John C.S Lui. "DroidTrace: A Ptrace Based Android Dynamic Analysis System with Forward Execution Capability". In: *IWCMC 2014 - 10th International Wireless Communications and Mobile Computing Conference*. Aug. 2014, pp. 128–133. DOI: 10.1109/IWCMC.2014.6906344.

[11] John Jenkins and Heipeng Cai. "ICC-Inspect: Supporting Runtime Inspection of Android Inter-Component Communications". In: *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. May 2018, pp. 80–83. DOI: 10.1145/3197231.3197233.

[12] Ke Xu, Yingjiu Li, and Robert Deng. "ICCDetector: ICC-Based Malware Detection on Android". In: *IEEE Transactions on Information Forensics and Security* 11 (June 2016). DOI: 10.1109/TIFS.2016.2523912.

[13] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. "DroidChameleon: Evaluating Android anti-malware against transformation attacks". In: *ASIA CCS 2013 - Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. May 2013, pp. 329–334. DOI: 10.1145/2484313.2484355.

[14] Yanick Fratantonio et al. "TriggerScope: Towards Detecting Logic Bombs in Android Applications". In: *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 377–396. DOI: 10.1109/SP.2016.30.

[15] Thanasis Petsas et al. “Rage against the virtual machine: hindering dynamic analysis of Android malware”. In: *Proceedings of the 7th European Workshop on System Security, EuroSec 2014*. Apr. 2014. DOI: 10.1145/2592791.2592796.

[16] Ben Gruver. *Smali/baksmali*. Mar. 3, 2021. URL: https://github.com/JesusFreke/smali (visited on 07/11/2021).

[17] Raja Vallée-Rai and L. Hendren. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. Jan. 2004.

[18] Connor Tumbleson and Ryszard Wiśniewski. *Apktool. A tool for reverse engineering Andriod apk files*. Dec. 2, 2020. URL: https://ibotpeaches.github.io/Apktool/ (visited on 07/12/2020).

[19] *Dalvik Bytecode*. Google. Sept. 1, 2020. URL: https://source.android.com/devices/tech/dalvik/dalvik-bytecode (visited on 04/24/2021).

[20] Christopher Will. “A Framework for Automated Instrumentation of Android Applications”. Bachelor’s Thesis. Oct. 15, 2013.

[21] Parvez Faruki et al. “Android Security: A Survey of Issues, Malware Penetration, and Defenses”. In: *IEEE Communications Surveys & Tutorials* 17.2 (2015), pp. 998–1022. DOI: 10.1109/COMST.2014.2386139.

[22] Aisha Ali-Gombe et al. “AspectDroid: Android App Analysis System”. In: *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. CODASPY ’16. New Orleans, Louisiana, USA: Association for Computing Machinery, Mar. 9, 2016, pp. 145–147. ISBN: 9781450339353. DOI: 10.1145/2857705.2857739.

[23] Haipeng Cai and Barbara G. Ryder. “DroidFax: A Toolkit for Systematic Characterization of Android Applications”. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Sept. 2017, pp. 643–647. DOI: 10.1109/ICSME.2017.35.

[24] Kun Yang. *apkil. An APK instrumentation library and DroidBox APIMonitor*. Apr. 23, 2013. URL: https://github.com/kelwin/apkil (visited on 07/18/2021).

[25] Patrik Lantz. *DroidBox. Dynamic analysis of Android apps*. Oct. 17, 2019. URL: https://github.com/pjlantz/droidbox (visited on 07/18/2021).

[26] Kun Yang. *Beta Release of DroidBox for Android 2.3 and APIMonitor*. URL: https://web.archive.org/web/20161219204143/https://www.honeynet.org/node/940 (visited on 07/18/2021).

[27] Oscar Somarriba et al. “Detection and Visualization of Android Malware Behavior”. In: *Journal of Electrical and Computer Engineering* 2016 (Mar. 2016), pp. 1–17. DOI: 10.1155/2016/8034967.

[28] Yan Hu et al. “Lightweight Energy Consumption Analysis and Prediction for Android Applications”. In: *Science of Computer Programming* 162 (May 2017), pp. 132–147. DOI: 10.1016/j.scico.2017.05.002.

[29] Lingzhi Qiu et al. “AppTrace: Dynamic trace on Android devices”. In: *2015 IEEE International Conference on Communications (ICC)*. June 2015, pp. 7145–7150. DOI: 10.1109/ICC.2015.7249466.

[30] Benjamin Davis et al. “I-arm-droid: A rewriting framework for in-app reference monitors for android applications”. In: *Proceedings of the Mobile Security Technologies 2012, MOST ’12. IEEE*. 2012.

[31] Jierui Liu et al. “InsDal: A safe and extensible instrumentation tool on Dalvik byte-code for Android applications”. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Feb. 2017, pp. 502–506. DOI: 10.1109/SANER.2017.7884662.

[32] Aleieldin Salem, F. Franziska Paulus, and Alexander Pretschner. “Repackman: A Tool for Automatic Repackaging of Android Apps”. In: *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*. A-Mobile 2018. Montpellier, France: Association for Computing Machinery, Sept. 4, 2018, pp. 25–28. ISBN: 9781450359733. DOI: 10.1145/3243218.3243224.

[33] Alexandre Bartel et al. “Improving Privacy on Android Smartphones Through In-Vivo Bytecode Instrumentation”. In: *ArXiv* abs/1208.4536 (2012).

[34] Jinseong Jeon et al. “Dr. Android and Mr. Hide: Fine-Grained Permissions in Android Applications”. In: *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM ’12. Raleigh, North Carolina, USA: Association for Computing Machinery, Oct. 2011, pp. 3–14. ISBN: 9781450316668. DOI: 10.1145/2381934.2381938.

[35] Rubin Xu, Hassen Saïdi, and Ross J. Anderson. “Aurasium: Practical Policy Enforcement for Android Applications”. In: *Proceedings of the 21st USENIX Conference on Security Symposium*. Security’12. Bellevue, WA: USENIX Association, 2012.

[36] Hao Hao, Vicky Singh, and Wenliang Du. “On the effectiveness of API-level access control using bytecode rewriting in Android”. In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ASIA CCS ’13. Hangzhou, China: Association for Computing Machinery, 2013, pp. 25–36. ISBN: 9781450317672. DOI: 10.1145/2484313.2484317.

[37] Pingfan Kong et al. “Automated Testing of Android Apps: A Systematic Literature Review”. In: *IEEE Transactions on Reliability* 68 (2019), pp. 45–66.

[38] Saswat Anand. *ELLA: A Tool for Binary Instrumentation of Android Apps*. May 26, 2016. URL: https://github.com/saswatanand/ella (visited on 07/14/2021).

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

[40] Chun-Ying Huang et al. “Code Coverage Measurement for Android Dynamic Analysis Tools”. In: *2015 IEEE International Conference on Mobile Services* (2015), pp. 209–216. DOI: 10.1109/MobServ.2015.38.

[41] Ferenc Horváth et al. “Code Coverage Measurement Framework for Android Devices”. In: *Acta Cybernetica* 21 (2014), pp. 439–458.

[42] Thomas Bläsing et al. “An Android Application Sandbox system for suspicious software detection”. In: *Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software, Malware 2010*. Nov. 2010, pp. 55–62. DOI: 10.1109/MALWARE.2010.5665792.

[43] Sven Bugiel et al. *XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks*. Tech. rep. 2011-01.

[44] Vaibhav Rastogi, Yan Chen, and William Enck. “AppsPlayground: Automatic Security Analysis of Smartphone Applications”. In: *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*. CODASPY ’13. San Antonio, Texas, USA: Association for Computing Machinery, Feb. 2013, pp. 209–220. ISBN: 9781450318907. DOI: 10.1145/2435349.2435379.

[45] Victor van der Veen. *Dynamic Analysis of Android Malware*. Aug. 2013. DOI: 10.13140/2.1.2373.4080.

[46] Yuan Zhang et al. "Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany: Association for Computing Machinery, Nov. 2013, pp. 611–622. ISBN: 9781450324779. DOI: 10.1145/2508859.2516689.

[47] William Enck et al. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *ACM Transactions on Computer Systems* 32.2 (June 2014). ISSN: 0734-2071. DOI: 10.1145/2619091.

[48] Kimberly Tam et al. "CopperDroid: Automatic Reconstruction of Android Malware Behaviors". In: *NDSS*. Jan. 2015. DOI: 10.14722/ndss.2015.23145.

[49] Aleksandr Pilgun. *ACVTool*. Oct. 12, 2020. URL: https://github.com/pilgun/acvtool (visited on 07/18/2021).

[50] Google. *Logcat command-line tool*. June 22, 2021. URL: https://developer.android.com/studio/command-line/logcat (visited on 07/19/2021).