Instrumentation blueprints: towards combining several android instrumentation tools

Staaij, A. van der; Gadyatskaya, O.; Zhou, J.; Adepu, S.; Alcaraz, C.; Batina, L.; ... ; Zonouz, S.

Instrumentation Blueprints: Towards Combining Several Android Instrumentation Tools

Arthur van der Staaij and Olga Gadyatskaya(✉)

LIACS, Leiden University, Leiden, The Netherlands
a.j.w.van.der.staaij@umail.leidenuniv.nl,
o.gadyatskaya@liacs.leidenuniv.nl

Abstract. The explosive growth of the amount of Android apps has given rise to a pressing need to analyse these apps, most importantly for security purposes. Many Android app analysis and hardening tools rely on *bytecode instrumentation*: the modification of the compiled app code. App instrumentation tools have all kinds of purposes, ranging from the measurement of code coverage to placing probes for malware detection. Given this variety, it may be useful to work with multiple tools that rely on instrumentation at the same time. The composition of such tools can however lead to issues, 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. Our evaluation shows that the proposed approach is viable.

Keywords: Android · App instrumentation · Instrumentation blueprints

1 Introduction

Over the last few years, the smartphones market has continued to grow. According to Statcounter, today Android is the most popular mobile operating system today, with a market share exceeding 70%[1]. Many apps are released for this operating systems every day: according to Statista, approximately 81,000 apps were released on Google Play only during February 2022[2]. As the size of the Android ecosystem grows, so does the demand for tools that analyze its apps. And indeed, in the recent years many such tools have become available.

[1] https://gs.statcounter.com/os-market-share/mobile/worldwide Data for February 2022; last accessed on March 18, 2022.

[2] https://www.statista.com/statistics/1020956/android-app-releases-worldwide/ Last accessed on March 18, 2022.

J. Zhou et al. (Eds.): ACNS 2022 Workshops, LNCS 13285, pp. 495–512, 2022.
https://doi.org/10.1007/978-3-031-16815-4_27

Android app 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 (e.g., [6–8,23,26,34]), a topic that has not been considered in detail in the community is the *composition* of instrumentation tools: using multiple such tools at once.

Indeed, given the variety of the available instrumentation tools, 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 complete the results are, and composing multiple app hardening tools to gain the benefits of each of them. To the best of our knowledge, not much research has been done in this area.

For dynamic 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 do not always behave in the same way (see, e.g., [23]), since they may use some form of external input like the internet or 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 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 it is known that success rate of individual tools is often much lower than 100% [23].

In order to facilitate the composition of instrumentation tools, this work 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 a dedicated *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.

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 proof-of-concept implementation of a blueprint output for ACV-Tool [23][3].

[3] Available at https://gitlab.com/avdstaaij-academic/citfaa-acvtool-fork.

4. The implementation of a prototype blueprint applicator program released open-source for the community to build on this work[4].
5. Evaluation of the prototypes on a case study.

2 Instrumentation Tools in the Literature

2.1 Taxonomy of Instrumentation Tools

We can distinguish two different types of app 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 [17], AspectDroid [2], DroidFax [7], APIMonitor [34] (a system that was used in a version of DroidBox [19,35]) and other unnamed tools [15,27,32]. AppTrace [24] is an example of a dynamic instrumentation tool.

The contributions of this work apply to only the static instrumentation approaches. From this point on, we will refer tools that use some form of static Dalvik bytecode instrumentation as simply *instrumentation tools*.

Instrumentation tools have a variety of purposes. DroidFax [7] and the tool developed by Somarriba et al. [27] monitor and visualize the runtime behavior of apps. ICCInspect [17] provides statistics and visualizations for the runtime usage of the Android ICC system. The tool from Hu et al. [15] 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*. AspectDroid [2], DroidBox [19] and the tool developed by Will [32] are examples of taint tracking tools that rely on app instrumentation.

There are also a number of frameworks for development of instrumentation tools. Examples are Apkil [34], I-ARM-Droid [8] and InsDal [20]. An interesting system that also somewhat fits in this category is Repackman [26], which can repackage apps with arbitrary payloads in order to evaluate other tools that detect such repackaging.

Some instrumentation tools 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 [4], describing the use cases of advertisement removal and the injection of a more fine-grained permission system. The system from [18] also involves bytecode instrumentation to make the Android permission system more fine-grained. Aurasium [33] is yet another example. An overview of techniques of this family is given by [13].

[4] https://gitlab.com/avdstaaij-academic/citfaa-blueprint-applicator.

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. Examples of instrumentation tools that measure code coverage are Ella [3], CovDroid [36], the tools described in [16] and [14], ACVTool [23], and COSMO [25]. ACVTool appears to be the most mature tool of this group, and [22,23] describe many more tools, alternative approaches and uses for code coverage measurements.

2.2 Limitations of Instrumentation

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 et al. report that instrumentation success rates (the fraction of apps that remains functional after instrumentation) of older code coverage tools lie between 36% and 65% [23]. ACVTool and COSMO have much higher success rates, but they cannot successfully instrument every app either.

As an alternative to bytecode instrumentation, many tools (e.g. [5,10,29]) 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.

To summarize, a rich variety of instrumentation tools exist in the literature, but, even though many of their goals are complimentary, to the best of our knowledge, nobody has investigated composing several tools. This is the gap that we start to address with this work.

3 Background

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.

Dalvik Bytecode and the Smali Representation. Dalvik bytecode is in many ways similar to machine code, consisting of low-level instructions like `add` and `goto` [9]. The full instruction list can be found in [1]. There are, however, some significant

Table 1. Example register layout (adapted from [32])

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

differences as well. For example, the Android Runtime is a *register-based* virtual machine [9]. 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 representation is too low-level, instrumentation tools usually do not deal with it directly. Instead, a human-readable assembly-like representation of it is used. There are two such representations that are commonly used: Smali [12] and Jimple [31]. 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.

In this work we use Apktool [30], which relies on `smali/baksmali`, to disassemble APKs into Smali files. A separate file is used for every Java class. Figure 1 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).

```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

Fig. 1. Example of a Java file and the corresponding Smali code

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. For the lack of space we only discuss how we approached the register management in this work (Sect. 5.2). The challenges of register management and an alternative solution are discussed in detail in [22,28].

4 Instrumentation Blueprints

In order to better facilitate the composition of instrumentation tools, we will now introduce our main contribution: the concept of *instrumentation blueprints*.

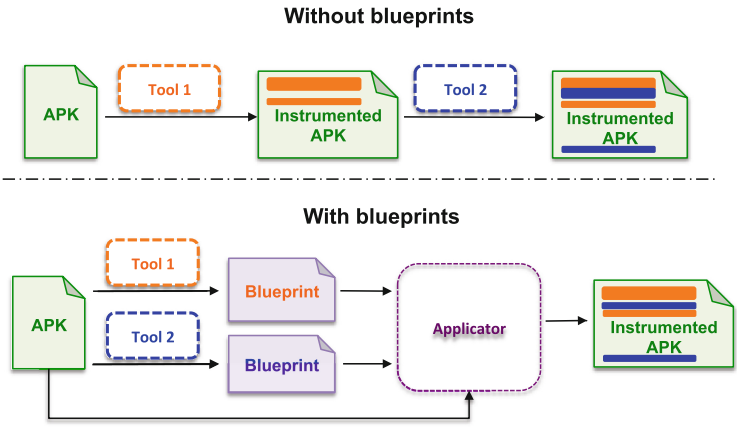

Fig. 2. Process flow for instrumentation composition with and without blueprints

As stated in Sect. 1, the main disadvantage of using instrumentation tools one after the other is that they will instrument each other's changes. 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. Figure 2 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. Since 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.

4.1 Blueprint Design

Practically, an instrumentation blueprint is a file that contains all changes that an instrumentation tool wants to apply to the bytecode. It is essentially a kind of `diff`, but a bit richer. 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 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 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. We believe that many 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 do not intend to change for the purpose of register management. For example, ACVTool needs to change every line that contains a parameter register [23]. 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 (described in Sect. 5.2).

4.2 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 Fig. 3. The `<method>` field specifies the fully qualified descriptor of the method, and the `<register-count>` field

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

(a) Method entry

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

(b) Line entry

Fig. 3. Formats of method and line entries

Table 2. Possible line entry operations

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

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 Fig. 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.

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 (1) empty lines; (2) the line containing `.locals` or `.registers`; (3) 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 Sect. 5.4. A concrete example of the blueprint syntax will be given in Sect. 5.1.

5 Implementation

We have implemented our blueprint system from two directions: we extended ACVTool [23] 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.

5.1 Generation of Instrumentation Blueprints for ACVTool

We have extended ACVTool to generate a blueprint as a side-effect of instrumentation[5]. 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 (discussed in Sect. 5.2).

ACVTool is written in Python and its source code is publicly available [21]. We refer the interested reader to [22] for the detailed explanation of the ACVTool instrumentation process. The tool uses a modified version of Apkil, a bytecode instrumentation library that was originally created for APIMonitor [34]. 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 the ACVTool code where Smali was inserted into the application and added blueprint generation code for each of them. ACVTool creates an auxiliary `.pickle` file, used to generate a report from the analysis results. We made ACVTool additionally create a blueprint file at the same location.

Figure 4 shows the blueprint segment for an ACVTool instrumentation example. The lines that are highlighted[6] in Fig. 4b which also appear in Fig. 4c 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. Since register management has been abstracted away by the `i`-register system, these two instructions should not be included in the blueprint.

[5] Our extension of ACVTool is available at https://gitlab.com/avdstaaij-academic/citfaa-acvtool-fork.

[6] Highlighted in yellow lines are those added by ACVTool during instrumentation.

```
.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 by ACVTool 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

Fig. 4. An example blueprint segment for ACVTool

5.2 Blueprint Applicator

Besides designing a prototype blueprint syntax and extending ACVTool to generate blueprints, we also created the prototype blueprint applicator program `applybp`[7]. 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

[7] https://gitlab.com/avdstaaij-academic/citfaa-blueprint-applicator.

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 Sect. 4.1, 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 5 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 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

```
@@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

Fig. 5. The result of merging two blueprints

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.

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.

Register Management Approach. To manage registers, `applybp` uses the same method as ACVTool [22] and the tool described by Will [32], because Pilgun has shown that this method is very robust [22]. 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 6 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 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.

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 straightforward: prepend contents are added before the line, append contents are added after, and if there is a replace entry,

```
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

Fig. 6. Register management process

the line is replaced with its contents. For every line of Smali 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 (Fig. 6c).

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` [11] 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.
- 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 Fig. 1b). 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 [1].

Our program is still a prototype, and as such, it still has a few limitations discussed in Sect. 5.4. Our prototype is released to the community[8].

[8] https://gitlab.com/avdstaaij-academic/citfaa-blueprint-applicator.

5.3 Evaluation

We tested the correctness of our ACVTool blueprint-generation extension and our blueprint application program `applybp` in a case study, using an app from F-droid[9]. For the case study, we used *Lesser Pad*, a simple note-taking app from F-droid[10]. We generated a blueprint for the it, applied it to the original app using `applybp` and checked whether the resulting app ran without problems on the Android Emulator. As a result, the app, which ran successfully with ACVTool's instrumentation, also did so after being instrumented according 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*. 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 obtained coverage reports were identical. Screenshots of the generated reports can be inspected in [28]. We note that the experiment was rather informal: we did not use a testing framework to repeat the exact same inputs for each version.

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 Sect. 5.2). 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.

5.4 Limitations

Although we believe that our blueprint composition approach is promising, it does have a number of limitations. Our blueprint system can only be used

[9] https://www.f-droid.org/.
[10] https://f-droid.org/en/packages/org.pulpdust.lesserpad/.

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). 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 Sect. 4.1. 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.

We already mentioned the limitations of our prototype blueprint syntax in Sect. 5.3. 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 limitation is the bad blueprint parsing performance. This could be improved by further optimizing the program or by redesigning blueprints to use multiple files.

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 yet have empirical evidence that shows whether our approach works for most tools, nor whether it actually improves the success rate of instrumentation composition. We plan to address this limitation in the future work.

6 Conclusions and Future Work

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 [23] to emit blueprints, and we have implemented the program `applybp` that can apply them. We have performed a case study showing that our approach can work in practice. Our proposed blueprint system may 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, improving the prototype will 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 in order. Finally, it would be interesting to explore automated generation of blueprints and to empirically assess challenging arising from combining multiple instrumentation tools.

Acknowledgements. We thank the anonymous reviewers for their useful comments.

References

1. Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode
2. Ali-Gombe, A., Ahmed, I., Richard, G.G., Roussev, V.: AspectDroid: Android app analysis system. In: Proceedings of the CODASPY, pp. 145–147 (2016)
3. Anand, S.: ELLA: a tool for binary instrumentation of Android apps. https://github.com/saswatanand/ella
4. Bartel, A., Klein, J., Martin, M., Allix, K., Traon, Y.L.: Improving privacy on Android smartphones through in-vivo bytecode instrumentation. arXiv abs/1208.4536 (2012)
5. Bläsing, T., Batyuk, L., Schmidt, A.D., Camtepe, S., Albayrak, S.: An Android application sandbox system for suspicious software detection. In: Proceedings of Malware, pp. 55–62 (2010)
6. Cai, H., Meng, N., Ryder, B., Yao, D.: DroidCat: effective Android malware detection and categorization via app-level profiling. IEEE Trans. Inf. Forensics Secur. **14**(6), 1455–1470 (2018)
7. Cai, H., Ryder, B.G.: DroidFax: a toolkit for systematic characterization of Android applications. In: Proceedings of ICSME, pp. 643–647 (2017)
8. Davis, B., Sanders, B., Khodaverdian, A., Chen, H.: I-ARM-Droid: a rewriting framework for in-app reference monitors for Android applications. Mob. Secur. Technol. **2012**(2), 1–7 (2012)
9. Ehringer, D.: The Dalvik virtual machine architecture **4**(8), 72 (2010). Technical report, March 2010
10. Enck, W., et al.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. **32**(2), 1–29 (2014)
11. Google: Logcat command-line tool. https://developer.android.com/studio/command-line/logcat
12. Gruver, B.: Smali/baksmali. https://github.com/JesusFreke/smali
13. Hao, H., Singh, V., Du, W.: On the effectiveness of API-level access control using bytecode rewriting in Android. In: Proceedings of ASIACCS, pp. 25–36 (2013)
14. Horváth, F., Bognár, S., Gergely, T., Rácz, R., Beszédes, Á., Marinkovic, V.: Code coverage measurement framework for Android devices. Acta Cybernet. **21**, 439–458 (2014)
15. Hu, Y., Yan, J., Yan, D., Lu, Q., Yan, J.: Lightweight energy consumption analysis and prediction for Android applications. Sci. Comput. Program. **162**, 132–147 (2017)
16. Huang, C.Y., Chiu, C.H., Lin, C., Tzeng, H.W.: Code coverage measurement for android dynamic analysis tools. In: Proceedings of MobServ, pp. 209–216 (2015)

17. Jenkins, J., Cai, H.: ICC-inspect: supporting runtime inspection of Android inter-component communications. In: Proceedings of MOBILESoft, pp. 80–83 (2018)
18. Jeon, J., et al.: Dr. Android and Mr. Hide: fine-grained permissions in Android applications. In: Proceedings of SPSM, pp. 3–14 (2012)
19. Lantz, P.: Droidbox. https://github.com/pjlantz/droidbox
20. Liu, J., Wu, T., Deng, X., Yan, J., Zhang, J.: InsDal: a safe and extensible instrumentation tool on Dalvik byte-code for Android applications. In: Proceedings of SANER, pp. 502–506 (2017)
21. Pilgun, A.: Acvtool. https://github.com/pilgun/acvtool
22. Pilgun, A.: Instruction coverage for Android app testing and tuning. Ph.D. dissertation, University of Luxembourg (2020)
23. Pilgun, A., Gadyatskaya, O., Zhauniarovich, Y., Dashevskyi, S., Kushniarou, A., Mauw, S.: Fine-grained code coverage measurement in automated black-box Android testing. ACM Trans. Softw. Eng. Methodol. (TOSEM) **29**(4), 1–35 (2020)
24. Qiu, L., Zhang, Z., Shen, Z., Sun, G.: AppTrace: dynamic trace on Android devices. In: Proceedings of ICC, pp. 7145–7150 (2015)
25. Romdhana, A., Ceccato, M., Georgiu, G.C., Merlo, A., Tonella, P.: COSMO: code coverage made easier for Android. In: Proceedings of ICST, pp. 417–423 (2021)
26. Salem, A., Paulus, F.F., Pretschner, A.: Repackman: a tool for automatic repackaging of Android apps. In: Proceedings of A-Mobile, p. 25–28 (2018)
27. Somarriba, O., Zurutuza, U., Uribeetxeberria, R., Delosières, L., Nadjm-Tehrani, S.: Detection and visualization of Android malware behavior. J. Electr. Comput. Eng. **2016**, 1–17 (2016)
28. van der Staaij, A.: Composing instrumentation tools for Android apps. Bachelor's thesis, Leiden University (2021). https://theses.liacs.nl/2176
29. Tam, K., Khan, S., Fattori, A., Cavallaro, L.: CopperDroid: automatic reconstruction of Android malware behaviors. In: Proceedings of NDSS (2015)
30. Tumbleson, C., Wiśniewski, R.: Apktool. https://ibotpeaches.github.io/Apktool/
31. Vallée-Rai, R., Hendren, L.: Jimple: simplifying Java bytecode for analyses and transformations (1998)
32. Will, C.: A framework for automated instrumentation of Android applications (2013)
33. Xu, R., Saïdi, H., Anderson, R.J.: Aurasium: practical policy enforcement for Android applications. In: Proceedings of USENIX Security (2012)
34. Yang, K.: apkil. https://github.com/kelwin/apkil
35. Yang, K.: Beta release of DroidBox for Android 2.3 and APIMonitor. https://web.archive.org/web/20161219204143/. https://www.honeynet.org/node/940
36. Yeh, C.C., Huang, S.K.: CovDroid: a black-box testing coverage system for Android. In: Proceedings of COMPSAC, vol. 3, pp. 447–452 (2015)