Empirical Software Engineering (2023) 28:48
https://doi.org/10.1007/s10664-022-10281-9

# Android decompiler performance on benign and malicious apps: an empirical study

**Ulf Kargén[1] · Noah Mauthe[2] · Nahid Shahmehri[1]**

Accepted: 19 December 2022 / Published online: 20 February 2023

**Abstract**
Decompilers are indispensable tools in Android malware analysis and app security auditing. Numerous academic works also employ an Android decompiler as the first step in a program analysis pipeline. In such settings, decompilation is frequently regarded as a "solved" problem, in that it is simply expected that source code can be accurately recovered from an app. On the other hand, it is known that, e.g, obfuscation can negatively impact a decompiler's effectiveness. Therefore, in order to better understand potential failure modes of, e.g., automated analysis pipelines involving decompilation, it is important to characterize the performance of decompilers on both benign and malicious apps. To this end, we have performed what is, to the best of our knowledge, the first large-scale study of Android decompilation failure rates, using three sets of apps; namely, 3,018 open-source apps, 13,601 apps crawled from Google Play, and an existing collection of 24,553 malware samples. In addition to the state-of-the-art Dalvik bytecode decompiler Jadx, we also studied the performance of three popular Java decompilers. Furthermore, this paper also presents the findings from a follow-up study on 54,945 malware apps, where we additionally performed an analysis of the reasons for decompilation failures. Our study revealed that decompilers generally have very low failure rates, and that few failures on benign apps appear to be related to obfuscation. On malware, however, obfuscation appears to be a more prominent cause of failures, although the vast majority of malicious apps could still be fully decompiled by an ensemble of decompilers.

Communicated by: Rick Kazman, Marouane Kessentini, Yuanfang Cai

This article belongs to the Topical Collection: *Special Issue on Software Analysis, Evolution and Reengineering (SANER) 2021*

✉ Ulf Kargén
ulf.kargen@liu.se

Noah Mauthe
noah.mauthe@cispa.de

Nahid Shahmehri
nahid.shahmehri@liu.se

[1] Linköping University, Linköping, Sweden

[2] CISPA, Saarbrücken, Germany

**Keywords** Android · Mobile apps · Decompilation · Obfuscation · Reverse engineering · Malware

## 1 Introduction

Decompilers, i.e., tools that can reconstruct the source code from a program binary, are ubiquitous aids when reverse-engineering malware or performing software security auditing. On the Android platform, decompilers are also used extensively to lift the *Dalvik* bytecode of Android apps into Java source code, prior to manual or automated inspection (Gamba et al. 2020; Chen et al. 2019; Shan et al. 2018; Tian et al. 2018; Pauck et al. 2018; Xue et al. 2017; Cen et al. 2015).

Compared to native-code decompilation, reconstructing source code from Dalvik bytecode is, generally speaking, significantly less challenging. This is because many of the hurdles of decompiling native machine code are eliminated due to the constraints placed on Dalvik bytecode by the Android system. For example, simply recovering a correct assembly-code listing from a native binary is an undecidable problem in itself, since code and data can be interspersed (Linn and Debray 2003). Moreover, as a consequence of the relatively loose structure of common file formats for executables, it can be challenging to identify all functions (as well as their bounds) in a native binary (Pang et al. 2021). Neither of these problems exist in the Dalvik bytecode setting, since all methods are encapsulated in a highly-structured container format called DEX (Dalvik Executable format). An additional challenge often faced when attempting to decompile native code is control-flow obfuscation, which is frequently used by both malware authors and legitimate software developers to prevent decompilation or disassembly (Roundy and Miller 2013; Junod et al. 2015). Similarly, several obfuscation techniques exist for the Java virtual machine (Chan and Yang 2004; Hou et al. 2006), which are able to prevent decompilation of Java bytecode back into legible source code. Such obfuscation techniques rely on introducing "fake" branches, which do not affect runtime semantics, but which prevent reconstruction of high-level control-flow constructs (e.g., by making the static control-flow graph irreducible) (Collberg et al. 1997). Applying such obfuscation techniques on Dalvik bytecode, however, is technically much more challenging (albeit not impossible), due to the so-called *register-type conflict* problem (Balachandran et al. 2016), which we briefly describe in Section 2. For this reason, it is more common to instead apply data-obfuscation techniques, such as identifier renaming or string encryption, to Android apps. Such techniques primarily aim to hide clues about program semantics from human analysts, rather than preventing decompilation per se.

As a consequence of the differences discussed above, the typical usage models of decompilers differ for native code and Dalvik bytecode: while it is generally recognized that native-code decompilers in many cases fail to reconstruct syntactically and semantically correct source code, and are therefore best used as an aid for manual reverse engineering, Dalvik bytecode decompilation is frequently regarded as a "solved" problem, where it can simply be expected that correct Java source code can be fully reconstructed from a DEX file. This sentiment is often reflected in the Android literature, where, for example, many works (Chen et al. 2019; Li et al. 2017; Wang et al. 2015; Gibler et al. 2012; Martín et al. 2017; Enck et al. 2011) use decompilation as the first step in an automated analysis pipeline. However, previous work (Harrand et al. 2019) has shown that decompilers for the Java virtual machine (JVM) frequently produce code with subtle syntactic or semantic errors, raising the concern that such problems also exist when decompiling Dalvik bytecode. While systems that use decompilation to extract features for approximate app similarity metrics (e.g., Li et al. (2017), Wang et al. (2015), Martín et al. (2017), and Cen et al. (2015)) might be able to tolerate minor correctness errors without critical degradation of functionality[1], another potential cause for concern is the *completeness* of decompilation results. For example, a small-scale preliminary study on 151 open-source Android apps by Jang et al. (2019) indicated that popular decompilers frequently fail altogether to decompile a significant portion of methods in an app. Such completeness errors could potentially be even more detrimental to the reliability of automated analysis methods than minor syntactic or semantic decompilation errors.

It is clear from the above discussion that both the *correctness* and *completeness* of Android decompilation must be studied further. In this work, we have focused on the latter. As such, the first and primary research question (**RQ1**) that we have sought to answer is *To what degree can we expect decompilers to successfully recover source code from Android apps?*

Moreover, while control-flow obfuscation is presumably more rarely encountered in Dalvik bytecode than in native code, due to the aforementioned register-type conflict problem, the question remains: *to what degree is decompilation-breaking obfuscation a concern when analyzing malware or commercial apps for the Android platform?* We address this as our second research question (**RQ2**).

Here, it should be noted that Android apps can also contain native code components, whose decompilation are subject to the same challenges as with other native-code binaries, and which can also be subjected to control-flow obfuscation. However, because the limitations of native-code decompilation has already been well-studied, and because of the very different usage model for native-code decompilation, we have chosen to limit our focus to Dalvik bytecode decompilation in this study.

Our third research question concerns the performance of individual decompilers. The study by Harrand et al. (2019) showed that various idiosyncrasies of JVM decompilers can cause significant differences in relative performance between decompilers, depending on the program being analyzed. In a follow-up study (Harrand et al. 2020), they also showed that decompilation results can be combined to improve the overall correctness of recovered code. Similarly, the small-scale study by Jang et al. (2019) indicated that the same also holds true for Android decompilers. To determine if these preliminary results can be generalized, we have sought to answer the question: *Do different Android decompilers tend to systematically fail on the same methods, or do their results complement each other?* (**RQ3**)

We have addressed the three research questions above in a previous study (Mauthe et al. 2021). In addition to providing an extended presentation of the findings from that study, this paper also presents the results from a follow-up study on a large set of Android malware samples. Since our original results indicated that many decompilation failures appeared to be caused by implementation-level deficiencies, rather than fundamental limitations of the decompilation algorithms, we wanted to further study the reasons why decompilers fail. Therefore, in addition to analyzing the new dataset in the context of our original research questions, we also introduced a fourth research question: *To what degree does implementation-level limitations, in contrast to fundamental algorithmic limitations, contribute to decompilation failures?* (**RQ4**) Below, we summarize the contributions of our original study, as well as the new contributions presented in this paper.

[1]For example, both Cen et al. (2015) and Martín et al. (2017), who use decompilation as part of their respective malware detection systems, mention explicitly that they don't expect recovered source code to be completely accurate.

### Original Contributions

- We have performed a large-scale study of the decompilation success rate (i.e., the ratio of methods for which the decompiler reports successful decompilation) for Android apps using four different decompilers. Our original evaluation was performed on three datasets, consisting of, respectively: 3,018 open-source apps from the F-Droid repository, 13,601 apps from a recent crawl of Google Play, and a collection of 24,553 Android malware samples collected between 2010 and 2016.
- We have characterized the differences in decompilation success rate between the datasets, and performed a preliminary analysis of potential causes of these differences.
- Furthermore, our statistical analysis was complemented with a manual analysis of a number of Android apps.

### New Contributions

- We recognized, as a threat to the validity of our original study, that many samples in the original malware set was quite old. Therefore, we have repeated the statistical analysis for research questions **RQ1**–**RQ3** on an additional large set of more up-to-date Android malware, consisting of 54,945 apps, and report on how the results differ from those of the old malware set.
- We have complemented the results for **RQ3** with a more in-depth analysis of decompiler co-failure rates.
- Finally, as the largest new contribution of this work, we have performed data-mining on all error messages emitted by decompilers, when run on the new malware dataset, in order to gain better insights into the reasons for decompilation failures (**RQ4**).

Additionally, we make our implementation and collected data available in the interest of open science[2].

The rest of the paper is structured in the following way: In Section 2, we provide some background on Android decompilation and obfuscation techniques. We outline the methodology for our study in Section 3, and present our results in Section 4. The results of our follow-up study on reasons for failures are presented in Section 5. We discuss the findings and potential threats to validity in Section 6, and survey related work in Section 7. Finally, Section 8 concludes the paper.

## 2 Background

In order to make the paper self-contained, we will start by providing some brief background information on a few important concepts.

**Android App Runtime Model** Android apps are developed in the Java or Kotlin languages, and compiled to Dalvik bytecode. Apps are distributed in the form of Android Application Packages (APKs), which contain one or more files of the DEX format. DEX files in turn contain a number of classes, including Dalvik bytecode for each method of a class. On Android versions prior to 5.0, Dalvik bytecode was interpreted by a virtual machine. Modern versions of Android instead use the Android Runtime (ART), which avoids the overhead

[2]https://github.com/NoahMauthe/decompilation_analysis

of interpretation by pre-compiling the Dalvik bytecode to native code when an app is first installed.

**Android Decompilation** In addition to native Dalvik decompilers[3], Java decompilers can often also be used on Android apps by first converting the Dalvik bytecode into equivalent bytecode for the JVM, using a tool such as ded (Enck et al. 2011) or dex2jar[4]. Since the Kotlin language is designed to be fully interoperable with Java, apps written in Kotlin can generally also be decompiled into Java source code.

**Android Obfuscation** Android apps frequently make use of obfuscation to prevent intellectual property theft, such as redistribution of paid apps, or ad-fraud. (The latter implies repackaging apps with modified identifier tokens for ad services, in order to gain ad revenue based on other developers' work.) One of the most common types of obfuscation is *identifier renaming*, wherein human-readable identifiers for, e.g., methods or variables, are replaced with meaningless strings. This obfuscation is sometimes also applied to open-source apps, since it tends to make the final APK smaller. Another common obfuscation method is string encryption, which works by removing strings from a DEX file and replacing them with an encrypted variant. Decryption routines are then injected at the places where strings are used in the code, so that the strings can be decrypted on-the-fly during runtime. A more advanced form of obfuscation is *class encryption*, whereby an entire class is stored in encrypted form and reconstructed at runtime using Java's reflection API. *Packing* is a similar approach to obfuscation, where all the Dalvik code of an app is stored in encrypted form, and decrypted at runtime using a wrapper program.

A common form of control-flow obfuscation works by inserting "fake" branches to random or invalid code locations, where the branches are guarded by so-called opaque predicates (see for example (Collberg et al. 1997; Linn and Debray 2003)). Such predicates are hard to evaluate statically, but always give the same outcome at runtime. This kind of obfuscation is applicable both to native code and to bytecode for the JVM, and provides a strong defense against decompilation, as it often cannot be automatically broken without resorting to prohibitively expensive methods, such as symbolic execution (Ming et al. 2015). On Android, however, this technique is considerably harder to implement due to the register-type conflict problem. While the JVM is stack based, the Dalvik virtual machine is register based. During compilation to native code, the ART compiler will check that there are no instances where a register holds data of conflicting types along any control-flow path in a method. (For example, if an integer is written to a register at some point, and at a later point that register is read as a floating point number, a register-type conflict is reported, and compilation is aborted.) "Fake" branches stemming from control-flow obfuscation frequently cause this type of conflict. While methods for partially overcoming this problem have been described by Balachandran et al. (2016), it is unclear to what degree, if any, this type of anti-decompilation technique is used in the wild for Dalvik bytecode.

[3]Note that we here use the term "native decompiler" to mean a decompiler designed specifically to decompile Dalvik bytecode, which is native to the Android system. This should not be confused with term "native *code* decompiler", which is a decompiler for machine code.

[4]https://github.com/pxb1988/dex2jar/

## 3 Methodology

In this section, we outline the methodology of our work. We begin with a detailed description of the approach used in our original study, followed by a discussion of some of its limitations. Finally, we describe the methodology used in our follow-up study.

### 3.1 Original Study

As depicted in Fig. 1, we begin by gathering APKs from three different sources, in order to study decompilation characteristics of different kinds of apps. We collected 3,018 open-source apps from the F-Droid repository[5] and 13,601 apps from the Google Play store. Finally, we used the existing Android Malware Dataset (AMD) compiled by by Wei et al. (2017), consisting of 24,553 Android malware samples collected between 2010 and 2016. While the samples in this dataset are quite old, a benefit of the AMD dataset is that each sample is labeled with its family, allowing us to compensate for bias due to some families being over-represented in the dataset.

#### 3.1.1 Gathering Apps

Retrieving apps from the F-Droid repository is quite straightforward, as all apps can simply be enumerated and downloaded. The Google Play store, however, does not allow downloading apps in bulk. Therefore, similarly to previous works, we had to implement a custom crawler by partially reverse-engineering the internal Google Play API. As our aim was to collect the most popular applications in the store (i.e., the ones with the largest user-exposure), we used an approach similar to, e.g., Backes et al. (2016) and crawled Google Play by category. Our crawler first retrieves the current set of thematic categories present in Google Play and then goes on to query each of those for their respective subcategories. These subcategories are not thematic, but instead are of a commercial nature, displaying the highest grossing, highest selling and most popular applications. As we only want to include free applications in our dataset[6], we omit crawling the highest selling applications and focus on the other two subcategories. The crawler then queries the store API for all applications contained in each subcategory, and downloads all of them. This way, our set of apps will consist of the most popular apps in each category.

As the top grossing categories may still contain paid apps and some applications are present in multiple subcategories, we needed to do further pruning of duplicates and apps that failed to download as we did not purchase them. After pruning, we ended up with the aforementioned number of unique apps from 34 categories.

#### 3.1.2 Measuring Decompiler Success Rate

In the next step, each app is decompiled with four different decompilers. In addition to the state-of-the art native Android decompiler Jadx[7], we also used the three popular Java

[5]https://www.f-droid.org/

[6]We deemed this a reasonable restriction, considering that only 3.9% of all Google Play apps were paid at the time we collected the apps, according to https://www.appbrain.com/stats/free-and-paid-android-applications.

[7]https://github.com/skylot/jadx

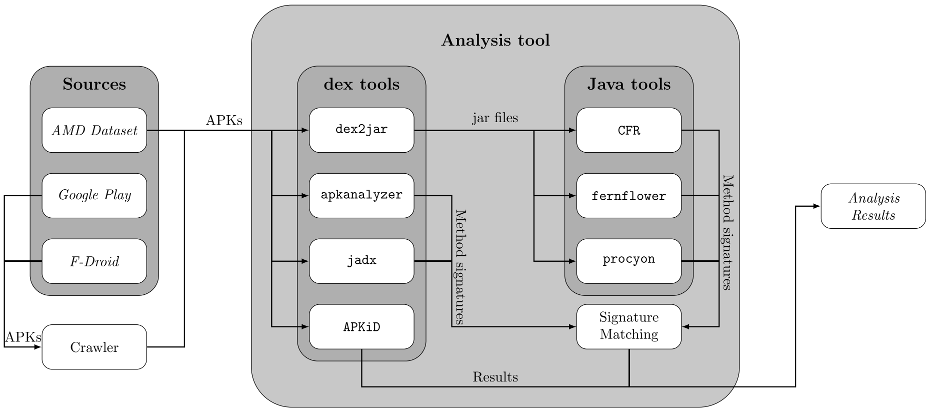

**Fig. 1** An overview of our analysis approach

decompilers CFR[8], Fernflower[9] and Procyon[10]. Before invoking the Java decompilers, we convert each app's Dalvik bytecode to JVM bytecode using dex2jar. In case of failures, the error messages from each decompiler are fed to a custom parser that records the methods that failed to decompile. When the analysis of one app is complete, all output artifacts, such as log files and decompiled source code, are discarded in order to avoid excessive disk usage. Since decompilation sometimes takes a very long time for some apps, it was necessary to implement timeouts. We used a timeout of 5 minutes for dex2jar, and also set the timeout for each decompiler to 5 minutes.

Since packing effectively hides an app's code from static analysis, decompilation is of little use for packed apps, unless the app is first unpacked by manual analysis. For this reason, we also wanted to detect if an app had been obfuscated with a packer. To this end, we use the APKiD tool[11], which can detect signatures of many popular packers.

In order to compare the per-method performance of the decompilers, the final step of our approach is to unify decompiler outputs. We first extract signatures for every method in an app, using apkanalyzer[12] from the Android SDK. We use this list of method signatures as a reference point, and match these signatures with the failed methods of each decompiler. The total number of methods per app, and the size of each method (i.e., the size of the method's bytecode) is also determined using apkanalyzer. Since all decompilers use slightly different formats for method signatures in their error reporting, we first preprocess the failed signatures to have a unified format. We also had to modify CFR somewhat, so that it outputs sufficient information about methods that it failed to decompile. Finally, we perform a simple textual matching of the unified signatures.

Our analysis platform was implemented in around 3,800 lines of Python. Crawling the datasets took about one week, and performing the analysis of all apps required around 4 weeks when running in parallel on three machines, each fitted with an 8-core Intel 9700K CPU.

[8]https://www.benf.org/other/cfr/
[9]https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine
[10]https://bitbucket.org/mstrobel/procyon/
[11]https://github.com/rednaga/APKiD
[12]https://developer.android.com/studio/command-line/apkanalyzer

### 3.1.3 Limitations

One general limitation of our approach is that we only match *failed* methods between the decompilers. In other words, we assume that a decompiler will always either successfully decompile a method, or emit an error message in a predictable format. If there are corner cases where this assumption does not hold, i.e., where decompilers silently “ignore” methods, we would not detect this as a failure, but would simply assume that the method (as reported by apkanalyzer) was successfully decompiled. The reason why we did not opt for the opposite approach of matching *successfully* decompiled methods is that this would be considerably more technically challenging, as it would require parsing the decompiled source code. Apart from substantially increasing the processing time required for each app, accurately recovering method signatures from the reconstructed source code could also potentially prove challenging, since the decompilation output would likely not always be fully compliant Java code.

There are also some problems that stem from limitations in the tools we use. These are summarized below.

**Challenging Java Language Features** The way decompilers handle some specific features of Java reduces the accuracy of our signature matching. *Inner classes* is one such case. While apkanalyzer reports the fully quantified names of inner classes, some of the decompilers only report the method name and containing source code file of a failed method in an inner class. Therefore, we are forced to over-simplify in these cases, and consider all methods of inner classes with the same name in one file as matching, by omitting the inner class quantifier reported by apkanalyzer. This sometimes leads to an over-approximation of failures, namely when there are methods in multiple inner classes whose signatures match a decompilation failure. For example, consider a class `A` with two inner classes `1` and `2` where all three classes define a method `void m(boolean)`. This might seem like an artificial case, but it often happens if classes `1` and `2` extend class `A`. In this case, `apkanalyzer` would output three different quantified method signatures:

```
A       void m(boolean)
A$1     void m(boolean)
A$2     void m(boolean)
```

However, two of the decompilers in our study, namely Fernflower and Procyon, would report the same signature `A void m(boolean)` for a failure in any of the three classes. When matching the failures using our simplification, this leads to three recorded failures instead of one. While this is not a problem when computing the overall failure rate of an app (since we know the total number of methods and failures), a method-by-method comparison of decompiler performance will inevitably suffer from some imprecision.

*Generics* also pose a problem for our signature matching. Some of the decompilers replace any generic they identify with `java.lang.Object`, whereas others leave the generic identifier unchanged, (e.g., `E`, `T`, `R` or `V`). This leads to mismatches between decompilers. An additional issue that further exacerbates the problem is that apkanalyzer *sometimes* manages to infer the type of a generic statically, while none of our decompilers have that ability. In contrast to the problem with inner classes, we cannot deal with this problem by over-approximation, since our text matching approach simply cannot determine whether an identifier is a generic’s denomination or a class name. For this reason, if a method using generics fails to decompile, the failure will not be recorded, and the method will be incorrectly reported as successfully decompiled. Similarly to the problem with inner classes, only method-by-method comparisons will be affected by this problem.

**Other Tool Limitations** During our experiments, we encountered several cases where dex2jar or apkanalyzer failed with an error message. (Presumably, this happens mostly for obfuscated apps). Since we use the method listing produced by apkanalyzer as a reference for unifying results, we simply excluded apps where apkanalyzer failed from the study. For apps where dex2jar failed, we could only record results for Jadx.

A more severe problem, which we discovered during our manual analysis of apps, is that these tools sometimes seemingly process an app successfully, while in fact producing an incorrect or incomplete result. apkanalyzer occasionally fails to include methods, or sometimes entire classes, in its output. Since we base our matching and unification approach on the output from apkanalyzer, this inevitably leads to a few methods being missed. We also discovered an undocumented failure mode of dex2jar. Apparently, in some cases when the tool cannot convert a method from Dalvik to JVM bytecode, it simply emits a "stub" method with the same signature as the original method, but where the body is replaced with a single `throw`-statement, throwing a custom exception. We discovered that dex2jar sometimes, but not consistently, emits a warning in its log file when this happens. Since the stub methods are likely much easier to decompile than the original method, this error presumably leads to false negatives in the reporting of decompilation failures for our three Java decompilers. Moreover, since both the exception type and the accompanying error message string differ from case to case, it is not possible to reliably detect the error in an automatic way. We only spotted this problem for one of the manually analyzed malware apps, which appeared to be heavily obfuscated. We describe this case in more detail in Section 4.6.

To estimate how much the above limitations influence the efficacy of our matching algorithm, we investigated the number of cases in which we either failed to match any method (due to the problem with generics), or where we had several matches (due to the problem with inner classes). In all of the 14,256,783 decompilation failures we encountered, there were 670,035 (5%) failures with no match, 349,585 (2%) methods with more than one match (3.83 matches per method on average), and 13,237,163 (93%) methods with exactly one match. Unfortunately, the number of cases in which apkanalyzer fails to report methods cannot be quantified with our currently implemented approach.

## 3.2 Follow-Up Study

One limitation of our original study is that the malware dataset was a few years old, and might not fully reflect the current Android malware landscape. For this reason, we have conducted a follow-up study on a more up-to-date dataset. Specifically, we have used a subset of apps in the AndroZoo dataset by Allix et al. (2016), which is a continuously updated set of Android applications, consisting of more than 16 million apps at the time of writing. The apps in our subset were selected based on the criteria that they had been flagged as malware by at least 30 antivirus products when submitted to VirusTotal[13] (i.e., about half of the available antivirus products at VirusTotal). This resulted in a set of 54,945 apps, which are, with very high likelihood, malicious. According to the VirusTotal scan date (provided with the AndroZoo dataset), the oldest app in our subset was added in 2012, and the newest in 2021, with the majority being added 2018 or 2019.

The AndroZoo apps were analyzed using the same methodology as outlined in the sections above. As our original study indicated that many decompilation failures appeared to be due to implementation-level limitations rather than fundamental algorithmic limitations,

[13] https://www.virustotal.com/

we additionally wanted to study the reasons for decompilation failures in more detail. To this end, we saved all the error messages emitted each time a decompiler failed to recover source code for a method. We considered several different data-mining techniques for clustering error messages into semantically meaningful groups. However, after observing that error messages typically included the name of the exception that caused the failure, we concluded that grouping based on the exception type would be a natural way to achieve a precise and meaningful clustering. We performed a regular expression search of all error messages and extracted words ending with “Error” or “Exception”. (We also verified that no error message mentioned more than one exception type.) In order to detect exception types not following the typical naming nomenclature, we also scanned the source code of each decompiler to find definitions of classes that extended a class ending with “Error” or “Exception”. This yielded two additional custom exception types, which were also included in the regular expression pattern.

For each decompiler, we furthermore classified exceptions as “anticipated” or “unexpected”. We elaborate on the classification criteria in Section 5, in conjunction with the presentation of the results of this part of the study.

## 4 Results

In this section, we present the results of our empirical study, beginning with our original work. In the interest of brevity, we also present the corresponding analysis results for the AndroZoo dataset alongside the results for the three datasets in the original study.

### 4.1 Basic Dataset Statistics

Table 1 shows some basic properties of our four app datasets. We see that quite a large number of malware apps could not be analyzed with apkanalyzer, while dex2jar instead fails on almost 400 apps from Google Play. As previously mentioned, the apps where apkanalyzer failed were excluded from the study.

As can be seen from the table, only about 100 apps were recognized by APKiD as having been packed in each of the Google Play and AMD datasets. None of the open-source apps were reported as packed. This is unsurprising, as open-source developers would have little incentive to obfuscate their code. For our new malware dataset from the AndroZoo collection, however, we see a proportionally much greater number of apps being detected as packed.

The number of timeouts for each dataset and decompiler are shown in Table 2. The native Dalvik decompiler Jadx performs the best with only 15 timeouts. CFR also performs well with only a few timed-out apps. Fernflower, on the other hand, experiences a very large

**Table 1** Dataset characteristics

| Dataset | Total | Failed apkanalyzer | Failed dex2jar | Processed | Packed (APKiD) |
|---|---|---|---|---|---|
| F-droid | 3,018 | 0 | 0 | 3,018 | 0 |
| Google | 13,601 | 7 | 394 | 13,594 | 127 |
| AMD | 24,553 | 1,220 | 33 | 23,333 | 131 |
| Androzoo | 54,945 | 1,087 | 15 | 53,843 | 2106 |

**Table 2** Timeout statistics for the 4 decompilers

| Dataset | CFR | Fernflower | Jadx | Procyon |
|---|---|---|---|---|
| F-droid | 1 | 164 | 0 | 37 |
| Google | 21 | 7652 | 4 | 1419 |
| AMD | 8 | 1955 | 1 | 130 |
| Androzoo | 12 | 4988 | 10 | 165 |

number of timeouts. On the Google Play dataset in particular, Fernflower stands out by timing out for more than half of the apps.

The inaccuracies introduced by the limitations described in Section 3.1.3 are broken down in Tables 3 and 4. While Fernflower and Procyon had many superfluous matches, Jadx and CFR were not affected by this problem. This is because Jadx and CFR (after our modifications) provide information about inner classes in their error messages.

On the other hand, a large number of the methods Jadx reported as failed were unmatchable due to the problem with handling Java generics. For example, more than one third of the failures on Google Play apps could not be matched to a corresponding method reported by apkanalyzer. As previously mentioned, however, these problems only affect the accuracy of method-wise comparisons.

## 4.2 Decompiler Performance

Here, we report on the performance of individual decompilers. Figure 2 shows the failure rate distributions of the three decompilers. In order to make a fair comparison, here we have only included cases where all decompilers actually produced any output. That is, we have excluded all apps where at least one decompiler timed out, as well as the apps where dex2jar failed. Table 5 shows the corresponding mean failure rate percentages. The last row shows the weighted average of all datasets (i.e., the mean of the dataset means). It is evident that Jadx outperforms the other (non-native) decompilers by a broad margin. The weighted average method failure rate is only around 0.04% for Jadx, which is almost two orders of magnitude lower than that of CFR and Fernflower. We can also see that all decompilers performed differently on different datasets, with most decompilers having a significantly higher mean failure rate on the malware datasets. We elaborate on this further in Section 4.4.

For completeness, Table 6 shows the failure rates when timed-out apps are included. These apps are considered as having a failure rate of 100%.

**Table 3** Percentage of reported failed methods that were superfluous matches (due to inner classes)

| Dataset | CFR | Fernflower | Jadx | Procyon |
|---|---|---|---|---|
| F-droid | 0.0 | 29.255 | 0.0 | 24.874 |
| Google | 0.0 | 13.650 | 0.0 | 14.274 |
| AMD | 0.0 | 16.905 | 0.0 | 22.154 |
| Androzoo | 0.0 | 8.699 | 0.0 | 6.677 |

**Table 4** Percentage of failed methods that were unmatchable (due to Java generics)

| Dataset | CFR | Fernflower | Jadx | Procyon |
|---|---|---|---|---|
| F-droid | 4.378 | 0.140 | 13.551 | 8.685 |
| Google | 6.443 | 0.331 | 34.030 | 8.212 |
| AMD | 1.586 | 0.036 | 2.649 | 1.670 |
| Androzoo | 0.915 | 0.035 | 0.277 | 2.816 |

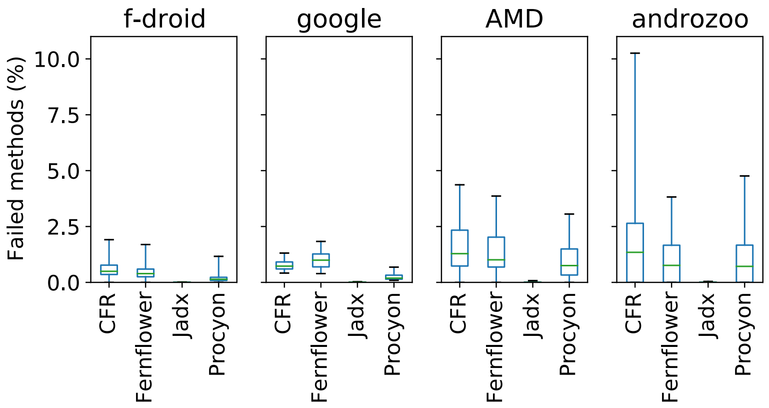

**Fig. 2** Failure rate distributions for the 4 decompilers, excluding timeouts and dex2jar failures. Whiskers show the upper and lower 5th percentiles

**Table 5** Mean failure rates in percent for the 4 decompilers, excluding timeouts and dex2jar failures

| Dataset | CFR | Fernflower | Jadx | Procyon |
|---|---|---|---|---|
| F-droid | 0.686 | 0.562 | 0.005 | 0.293 |
| Google | 0.844 | 1.051 | 0.011 | 0.321 |
| AMD | 1.751 | 1.459 | 0.047 | 1.078 |
| Androzoo | 2.964 | 1.154 | 0.104 | 1.546 |
| Weighted avg. | 1.561 | 1.057 | 0.042 | 0.809 |

**Table 6** Mean failure rates in percent for the 4 decompilers, including timeouts, but excluding dex2jar failures

| Dataset | CFR | Fernflower | Jadx | Procyon |
|---|---|---|---|---|
| F-droid | 0.741 | 5.966 | 0.005 | 1.522 |
| Google | 1.032 | 58.418 | 0.019 | 11.006 |
| AMD | 1.838 | 9.726 | 0.044 | 1.672 |
| Androzoo | 3.011 | 10.311 | 0.119 | 1.867 |
| Weighted avg. | 1.656 | 21.105 | 0.047 | 4.017 |

### 4.3 Failure Rate Diversity

In this section we explore the failure rate diversity of the 4 decompilers, i.e., the degree to which they complement each other in terms of successfully decompiling methods.

Table 7 shows the percentage of apps that could be fully decompiled, i.e., where the decompiler did not time out or experience other errors, and where no method decompilation failures were reported. Using Jadx alone (column 2), it was possible to fully decompile about 75% of the open-source apps, while only 21% of Google Play apps could be fully decompiled. This is probably due in part to Google Play apps having a much larger mean number of methods (63,748 methods on average for Google Play apps, versus 13,532 for F-Droid apps). Interestingly, around 80% of the apps in both the malware datasets could also be fully decompiled by Jadx. The fact that the malware apps had significantly fewer methods on average (5,142 and 3,667 for AMD and AndroZoo, respectively), compared to the other datasets, could partially explain this. (It should be noted, however, that many of the excluded apps for which apkanalyzer failed would probably also fail to decompile completely. These apps comprised about 5% and 2% of the respective datasets.) For completeness, Table 8 shows the corresponding figures also for the other three decompilers. Clearly, Jadx outperforms the other decompilers also in terms of how many apps that can be fully decompiled.

The next column in Table 7 shows the number of apps that could be fully decompiled by combining the results from all decompilers[14]. We see that, even with an ensemble, it was not possible to fully decompile all apps in any of the datasets. However, the ensemble improved the success rate significantly, especially for the Google Play dataset. The last column shows the percentage of apps that could be fully decompiled by all decompilers. These figures are negligible for the Google Play and AMD datasets, but, interestingly, quite high for the new AndroZoo dataset.

Another way to characterize the diversity of decompilers is to measure their co-failure rate. It should be noted that, since here we have to make comparisons between decompilers on a method-by-method basis, the aforementioned method matching limitations will influence the results. For this reason, the figures presented here must be taken as indicative, rather than exact. Furthermore, the large number of timeouts for some decompilers complicates the analysis of the co-failure rate. Therefore, we have chosen to include co-failure rates both for the case when timeouts are treated as failures[15], as well as the case when only actual failures are considered.

Table 9 shows the co-failure percentages for each decompiler, on each of the datasets. The table should be interpreted in the following way: One row shows, for the corresponding decompiler, out of the methods for which that decompiler failed, the percentage of those methods that the other decompilers also failed on. For example, we see from the part of the table that includes timeouts that, out of all methods that Jadx failed to decompile in the F-droid dataset, Fernflower also failed to decompile 30.89% of those methods. (Note that the co-failure relation is not symmetric.)

We can observe from the table that the co-failure rates are typically quite modest, explaining the improved performance of the ensemble. (The low co-failure rates of other

[14]In our original publication, there was an error in the calculation that made this number appear smaller than it really was. This has been corrected here, and some of our conclusions at the end of the paper have been updated accordingly.

[15]I.e., all methods are treated as failed in apps that the decompiler timed out on.

**Table 7** Percent of all apps where all methods were successfully decompiled by, respectively, Jadx, an ensemble of all decompilers, and individually by all decompilers

| Dataset | Jadx | Ensemble | All decompilers |
|---|---|---|---|
| F-droid | 74.52 | 98.31 | 8.65 |
| Google | 21.02 | 89.97 | 0.15 |
| AMD | 78.81 | 85.91 | 0.60 |
| Androzoo | 84.97 | 96.15 | 16.00 |

decompilers compared to Jadx are explained by the much lower overall failure rate of Jadx.) We can also observe that excluding timeouts affects the co-failure rates significantly. It is especially interesting to study this phenomenon for CFR and Jadx, who themselves have very low timeout rates. The large decrease in co-failures when timeouts are excluded could indicate that many error-conditions that CFR and Jadx handle "gracefully" (i.e., by reporting an error and aborting decompilation for the affected method), instead cause excessive computation times in the other two decompilers.

In order to better understand how common it is for several decompilers to fail for a given method, Table 10 shows the multi-co-failure percentage. Here, we have only considered the F-Droid dataset, since we expect the (presumably non-obfuscated) open-source apps to be less likely to trigger the undocumented failure mode of dex2jar that we describe in Section 3.1.3. For each decompiler, the tables shows the percentage of cases where, respectively, at least 1, 2 or 3 (i.e., all) other decompilers also failed on a method that the decompiler in question failed to decompile. (Here we also consider timeouts as failures.) For example, in 72% of cases where Jadx fails to decompile a method, at least one other decompiler also fails on that method. This figure is lower for all other decompilers, which can be explained by their overall higher failure rates.

An interesting finding is that, despite Jadx drastically outperforming the other decompilers, in about 96% of cases where Jadx fails to decompile a method, at least one other decompiler succeeds.

## 4.4 Differences Between Datasets

When investigating differences between the datasets in more detail, we choose to use only results from the native Jadx decompiler, as it provides the most comprehensive coverage of apps and generally outperforms the other decompilers. Figure 3 shows the mean Jadx failure rates for the datasets. Here, we have included those cases where dex2jar failed. However, this only marginally changes the results compared to those shown in Table 5. We note that the mean failure rate of Google Play apps is roughly twice that of the open-source apps,

**Table 8** Percent of all apps where all methods were successfully decompiled by the non-native decompilers

| Dataset | CFR | Fernflower | Procyon |
|---|---|---|---|
| F-droid | 11.20 | 13.39 | 14.61 |
| Google | 0.20 | 0.21 | 0.38 |
| AMD | 7.04 | 6.01 | 6.51 |
| Androzoo | 23.10 | 23.21 | 23.15 |

**Table 9** Co-failure percentages for decompilers on the different datasets

| | CFR | Fernflower | Jadx | Procyon | |
|---|---|---|---|---|---|
| | | Including timeouts | | | |
| CFR | 100 | 35.47 | 0.51 | 16.69 | f-droid |
| Fernflower | 1.34 | 100 | 0.01 | 7.98 | |
| Jadx | 58.74 | 30.89 | 100 | 8.62 | |
| Procyon | 3.21 | 40.55 | 0.01 | 100 | |
| CFR | 100 | 98.67 | 0.26 | 94.36 | google |
| Fernflower | 13.11 | 100 | 0.04 | 26.83 | |
| Jadx | 77.11 | 81.67 | 100 | 71.52 | |
| Procyon | 42.03 | 89.92 | 0.11 | 100 | |
| CFR | 100 | 53.71 | 1.24 | 56.59 | AMD |
| Fernflower | 5.97 | 100 | 0.14 | 6.19 | |
| Jadx | 86.51 | 86.34 | 100 | 80.96 | |
| Procyon | 40.84 | 40.25 | 0.84 | 100 | |
| CFR | 100 | 50.45 | 0.23 | 42.73 | androzoo |
| Fernflower | 3.54 | 100 | 0.05 | 3.51 | |
| Jadx | 7.04 | 22.32 | 100 | 11.89 | |
| Procyon | 33.47 | 39.19 | 0.31 | 100 | |
| | | Excluding timeouts | | | |
| CFR | 100 | 17.07 | 0.53 | 12.35 | f-droid |
| Fernflower | 18.70 | 100 | 0.05 | 11.47 | |
| Jadx | 58.69 | 5.35 | 100 | 3.09 | |
| Procyon | 37.07 | 31.41 | 0.08 | 100 | |
| CFR | 100 | 7.12 | 0.43 | 13.71 | google |
| Fernflower | 17.47 | 100 | 0.03 | 10.97 | |
| Jadx | 40.58 | 1.00 | 100 | 0.90 | |
| Procyon | 47.85 | 15.61 | 0.03 | 100 | |
| CFR | 100 | 28.41 | 0.38 | 36.96 | AMD |
| Fernflower | 31.08 | 100 | 0.24 | 32.79 | |
| Jadx | 58.04 | 33.27 | 100 | 40.26 | |
| Procyon | 52.66 | 42.70 | 0.38 | 100 | |
| CFR | 100 | 22.26 | 0.23 | 39.11 | androzoo |
| Fernflower | 39.84 | 100 | 0.15 | 37.66 | |
| Jadx | 7.99 | 2.90 | 100 | 3.41 | |
| Procyon | 66.69 | 35.88 | 0.17 | 100 | |

while for the AMD apps, the mean failure rate is roughly one order of magnitude higher. The malware in the new AndroZoo dataset shows even higher failure rates.

As seen above in Fig. 2, the failure rates vary significantly between different apps. Therefore, sampling error might be a concern when characterizing differences between the

**Table 10** Decompiler multi-co-failure percentages on the F-Droid dataset

| Decompiler | >0 other failed | >1 other failed | All other failed |
|---|---|---|---|
| CFR | 40.9549 | 11.6763 | 0.0378 |
| Fernflower | 8.8871 | 0.4419 | 0.0014 |
| Jadx | 71.9225 | 21.9917 | 4.3338 |
| Procyon | 41.5437 | 2.2230 | 0.0073 |

datasets. In order to quantify the effect of sampling error, we have computed 95% confidence intervals (shown as error bars in Fig. 3), using bootstrap sampling over all three datasets. We used 1,000 resamplings for our computations. As can be seen from the figure, there is a statistically significant difference between all four datasets.

Another potential concern is that the distribution of samples over different malware families is highly skewed in the AMD set of malicious apps (since we do not know the families for the AndroZoo malware, we cannot tell if such a skew exists also in that dataset). For example, around one third of the AMD apps belong to the same family. This could introduce bias in our results, since members of the same malware family are often highly similar. For this reason, we have also included a weighted mean, which is computed by taking the mean of the family-wise mean failure rate. As can be seen from the figure, bootstrapping over the family means revealed a very large variation in decompilation failure rate between different malware families.

We also investigated the differences between Google Play apps with and without ads (according to the Play Store metadata), and found that apps with ads had roughly 50% more decompilation failures on average. Specifically, apps with ads had a mean failure rate of 0.0135%, while the same figure for non-ad-supported apps was 0.00861%. Bootstrap sampling revealed that the difference was statistically significant.

We similarly compared mean failure rates for apps that were recognized as packed by APKiD, compared to the other apps. For Google Play, there was a statistically significant difference, with packed apps having 0.126% failures on average, compared to 0.0104% (a factor of 12) for non-packed apps. We also observed a similar statistically significant difference for the AndroZoo malware: 0.854% for packed apps versus 0.0694% for non-packed apps (also a factor 12). For the AMD malware, the corresponding figures were 0.154% and 0.0436%, respectively. This difference was not statistically significant, however. Since the wrapper code of many packers is often heavily obfuscated to frustrate manual unpacking,

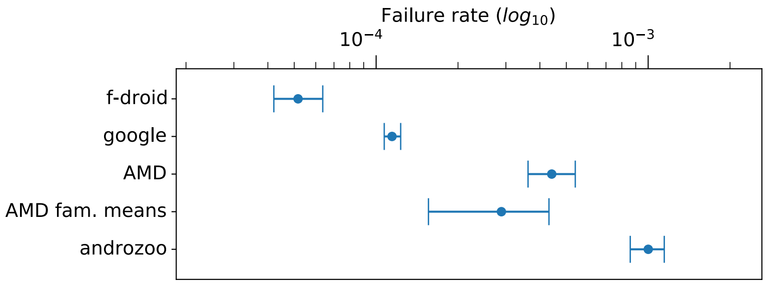

**Fig. 3** 95% confidence intervals for Jadx mean failure rates

we expected the figures to be higher for packed apps, compared to other apps. However, we were surprised to find that almost all methods in packed apps could often be decompiled.

## 4.5 Exploring Reasons for Differences

Here, we attempt to shed some light on the underlying reasons for the observed differences between the three datasets. As the results in this section required analysis at the granularity of individual methods, the aforementioned method-matching limitations also apply here.

Our primary hypothesis to explain the differences between datasets was that they exhibited differing prevalence of obfuscation. However, as preliminary analyses indicated that the likelihood of decompilation success depended on the size of a method's byte code, we wanted to rule out the alternative hypothesis that the differences were simply due to different method-size distributions. To this end, we divided all methods based on their size into logarithmically-spaced bins, and investigated the per-bin failure rates. The upper part of Fig. 4 shows the results. Especially for the benign apps, a strong, roughly linear dependence between method size and failure rate is evident in the log-log scale bar chart. The

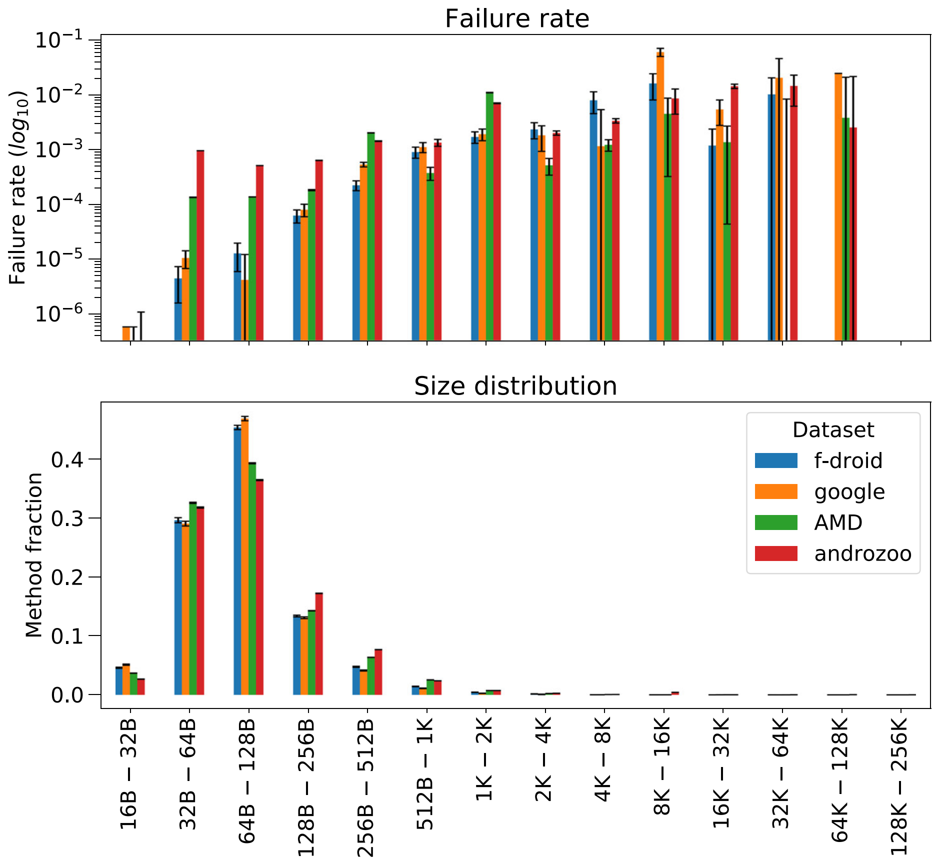

**Fig. 4** Jadx failure rate as a function of binned method sizes (top), and the distribution of method sizes, using the same bins (bottom)

failure rate of methods in the 8–16 kB bin is, for example, more than three orders of magnitude higher than for small methods in the 32–64 B bin. The error bars are again computed by 1,000-fold bootstrap sampling, and show the 95% confidence intervals. Since methods of several kB or more are very rare, the confidence intervals are generally very wide for the corresponding bins.

The method size distributions for the datasets are shown in the lower part of Fig. 4. Here, we see that the distributions are quite similar for all three datasets. In particular, we see that for the two most common method size intervals (32–64 B and 64–128 B), comprising around 70% of all methods, the failure rates of the AMD dataset is about one order of magnitude higher compared to the other datasets, while the AndroZoo malware has almost two orders of magnitude higher failure rates. This corresponds well with the results shown in Fig. 3. This suggests that the differences cannot be explained by different method size distributions.

For our last analysis, we wanted to make an exploratory study of the class names associated with frequent decompilation failures. For each method reported by apkanalyzer, we extracted the fully qualified name of the containing class, i.e., the package and class names. We then divided the string into tokens by splitting on the "`.`" (period) symbol. For each token, the number of method signatures in which the token appeared was recorded separately for each dataset, along with the percentage of those method occurrences that Jadx failed to decompile. Since we were interested in tokens associated with many failures, we filtered out tokens with less than 1% associated failure rates. Finally, we sorted the tokens on the total number of (method) occurrences, and picked the top 20 tokens for each dataset. Tables 11 and 12 show the results for the benign and malicious apps, respectively.

Several interesting patterns can be identified. We see that `SlidingWindowKt` and `windowedIterator`, which are both class names from the Kotlin standard library, are associated with a large number of failures in both the F-Droid and Google Play datasets. Since the Kotlin standard library is open source, it is unlikely to be obfuscated. Instead, this finding might suggest that Jadx is less effective at decompiling some bytecode compiled from Kotlin source code.

`ReaderBasedJsonParser` and `NonBlockingJsonParser`, which are names from the open-source Jackson parsing library, are also among the top 5 most failure-prone tokens in both F-Droid and Google Play apps. Similarly, the tokens `JSONLexerBase` and `JSONLexer` from another JSON parsing library are among the top 20 for Google Play. We also see several names associated with parsing of various data formats in the top 20 for the AMD dataset (`ZLDTDParser`, `ReaderBasedParser`, `Utf8StreamParser`, `WbxmlParser`). Similarly, several tokens associated with cryptography or encoding, or with known crypto libraries, are present in the top 20 for all datasets (`ASN1Set`, `ASN1Object`, `ConstructedOctetStream`, `DSAParametersGenerator`, `Encoder`, `base64`). This suggests that the decompiler has difficulties handling methods containing large chunks of code with complex computations and/or control flow, which are common in both parsing and cryptographic code, and that this type of code is a major contributor to decompilation failures.

The above findings indicate that a major part of the decompilation failures observed in our study are not due to deliberate attempts at preventing static analysis, but simply due to limitations of the decompilers. However, we also observed several tokens that appear to be associated with obfuscation. The Apptimize library, which is at the top of the list for Google Play, was found to be heavily obfuscated during our manual analysis (see Section 4.6). The

**Table 11** Top 20 class/package identifier tokens associated with Jadx decompilation failures for benign apps

| F-droid | | | Google | | |
|---|---|---|---|---|---|
| Token | Failures | Frequency | Token | Failures | Frequency |
| WindowedIterator | 183 | 20.29% | apptimize | 2595 | 1.72% |
| SlidingWindowKt | 183 | 9.30% | SlidingWindowKt | 1901 | 9.14% |
| ReaderBasedJsonParser | 83 | 1.56% | windowedIterator | 1863 | 19.49% |
| NonBlockingJsonParser | 48 | 3.52% | ReaderBasedJsonParser | 1390 | 1.60% |
| MergerBiFunction | 37 | 27.41% | NonBlockingJsonParser | 1112 | 3.48% |
| ASN1Set | 28 | 1.48% | BaseListBitmapDataSubscriber | 849 | 30.53% |
| ConstructedOctetStream | 28 | 12.17% | zzdfh | 772 | 5.54% |
| ConverterSet | 28 | 6.22% | MergerBiFunction | 664 | 22.13% |
| FixedPeriodTicker | 25 | 7.91% | ConverterSet | 553 | 8.42% |
| FlowKt__DelayKt | 25 | 1.24% | zzdph | 485 | 2.24% |
| Fx | 23 | 3.01% | zzdbm | 446 | 2.69% |
| InterruptibleTask | 22 | 4.12% | zzdme | 422 | 3.67% |
| LDAPStoreHelper | 19 | 1.44% | zzdha | 332 | 1.05% |
| X509LDAPCertStoreSpi | 19 | 3.63% | JSONLexerBase | 290 | 3.37% |
| BaseListBitmapDataSubscriber | 18 | 31.58% | InterruptibleTask | 283 | 3.09% |
| AbstractListeningExecutorService | 14 | 1.88% | ASN1Set | 241 | 2.31% |
| DSAParametersGenerator | 12 | 1.46% | MethodWriter | 236 | 1.31% |
| TokenStream | 11 | 1.14% | JSONLexer | 205 | 2.52% |
| ASN1Object | 10 | 1.69% | fixedPeriodTicker | 198 | 6.32% |
| NioClientManager | 10 | 7.69% | zzflf | 160 | 38.37% |

2,595 failed methods attributed to the library constitute around 3% of all observed Google Play failures. We also noted a number of tokens that seemed to be the result of identifier renaming (“`zzdfh`”, etc.) in the Google Play dataset. Since several of these tokens have a high associated failure rate, we speculate that they stem from code that has been subjected to some form of control-flow obfuscation, in addition to the identifier renaming.

Another third-party library, which appears to be a large sole contributor to decompilation failures in the AMD dataset, is BugSense. The `BugSenseHandler` token is associated with 2,034 failures, or about 21% of all failed methods in the dataset. Since this library is open-source, it is unlikely that it is distributed in obfuscated form. Instead, it seems that some of the code in this library is simply difficult to decompile.

The results for the AndroZoo dataset stand out from the other datasets by only having a single token associated with encoding, decoding, or parsing. Instead, many tokens appear related to various electronic-payment services, particularly for the Chinese market. Also, the `roothelper` token (the name of a library for performing privileged operations on rooted Android devices) have a very high failure-frequency. More than 95% of methods whose name included the token failed to decompile. These results more strongly point towards obfuscation being a factor for failures, since there is nothing about most of the tokens (comparing with the other datasets) that would suggest that the corresponding methods would be inherently harder-than-average to decompile.

**Table 12** Top 20 class/package identifier tokens associated with Jadx decompilation failures for malicious apps

| AMD | | | Androzoo | | |
|---|---|---|---|---|---|
| Token | Failures | Frequency | Token | Failures | Frequency |
| BugSenseHandler | 2034 | 1.50% | upay | 17016 | 1.82% |
| Encoder | 544 | 2.33% | billing | 13727 | 1.46% |
| ZLDTDParser | 438 | 50.00% | heju | 8863 | 4.31% |
| ReaderBasedParser | 426 | 3.30% | umpay | 5141 | 3.34% |
| Jianmo | 283 | 1.07% | huafei | 4828 | 2.67% |
| Igexin | 251 | 1.24% | huafubao | 4118 | 2.74% |
| Utf8StreamParser | 216 | 1.02% | anrd | 1906 | 3.58% |
| Imobile | 142 | 1.75% | Encoder | 1867 | 6.17% |
| Base64 | 107 | 1.63% | upay_sms | 1630 | 2.11% |
| Provider | 98 | 2.38% | html | 1563 | 5.27% |
| WbxmlParser | 60 | 1.15% | tdx | 1548 | 93.65% |
| SDK | 52 | 2.10% | roothelper | 1548 | 95.50% |
| Products | 52 | 10.18% | sysservices_t | 1393 | 2.66% |
| Inigma_sdk | 52 | 10.18% | tk | 1275 | 5.70% |
| Threegvision | 52 | 10.18% | system | 1215 | 1.17% |
| Rc | 49 | 3.38% | Silen | 825 | 27.17% |
| QueueDetails | 49 | 7.40% | ck_mdo | 759 | 4.21% |
| QueueOverview | 49 | 5.94% | he_danji | 735 | 2.49% |
| FaultTolerantNegotiator | 48 | 7.10% | yinlian | 734 | 1.20% |
| Qqmagic | 42 | 3.20% | tenpay | 725 | 2.27% |

## 4.6 Manual Analysis

In this section we describe the complementary manual analysis performed on the F-Droid, Google Play and AMD datasets in our original study.

For the analysis, we selected the 5 apps with the highest Jadx failure rate from each of the F-Droid and Google Play datasets. For the AMD dataset, we instead picked the sample with the highest failure rates from each family, and then selected the top 5 within this list. We used this approach in order to avoid potentially getting 5 very similar samples from the same family. Also, as decompilation is of little use for packed apps (since only the wrapper code can be decompiled), we omitted apps that were flagged as packed by APKiD.

We performed a detailed analysis of 10–20 methods in each app by comparing the output from decompilation (in cases where at least one decompiler succeeded) with the corresponding Dalvik bytecode, which was disassembled using baksmali[16]. In cases where all decompilers failed, we attempted to manually reverse-engineer the method from the bytecode. When necessary, we also made a more cursory investigation of other methods and classes. Methods were prioritized based on the number of failing decompilers. For apps with many failed methods, we took a random subset of methods where more than two decompilers failed. If an app had only a small number of failed methods (this was the case for the

[16]https://github.com/JesusFreke/smali

F-Droid apps), we picked the methods that had the largest number of failing decompilers. During the analysis, we attempted to investigate causes of decompilation failures, and also specifically looked for signs of obfuscation. The results for each dataset are summarized below.

**F-Droid** We found no evidence of obfuscation in any of the open-source apps. The failures we investigated appeared to be caused by very deep levels of nesting, and by complex control flow. In two apps, failures appeared to be caused by methods declared in anonymous inner classes, nested within several levels of other anonymous inner classes. In the three other apps, failures were caused by complex control flow inside `switch-case` constructs.

**Google Play** In four of the Google Play apps, we discovered that the decompilation failures were due to the third-party library Apptimize, which we mentioned above. The library is obfuscated by moving most of the logic of each class into a large static block. The control flow of the static blocks is highly complex, with many nested loops containing `break` statements that appear to be protected by opaque predicates. We also found at least one case of dead code insertion. Jadx reports the same error for all of these static blocks: “JADX OVERFLOW ERROR: regions count limit reached”.

The fifth app was also obfuscated, using a weak form of opaque predicates and excessive variable reassignments. In contrast to the Apptimize library, however, only a subset of the methods appeared to be obfuscated.

**AMD** All five malware apps were obfuscated with identifier renaming. However, obfuscation appeared to be the cause of decompilation failures for only one of the apps. This app had a Jadx failure rate of 63%, the highest among all apps across the three datasets. The other decompilers, however, reported much lower failure rates. This led us to discover the undocumented failure mode of dex2jar that we describe in Section 3.1.3. The failures appeared to be caused by a particularly intrusive form of obfuscation, which caused baksmali to crash due to unrecognized opcodes. We believe that the application may use an internal translation layer and altered bytecode that is only translated at runtime[17].

One of the most prominent causes of decompilation failures among the other four samples was excessive use of *try-catch* blocks for I/O or network error handling. We also found that decompilers often failed on conditionals that could be represented as ternary if-statements (i.e., conditionals that were translated to ternary if-statements by the non-failing decompilers).

## 5 Reasons for Failures

Both the statistical and manual analyses performed in our original study indicated that most decompilation failures were caused by imperfections in the decompiler tools, rather than by obfuscation. In most cases where we observed failures, the decompiler would emit error messages suggesting that the cause was some kind of internal resource-exhaustion (e.g., hitting some internal “limit”). This was often caused by very complex control flow, or very deep nesting levels of various kinds (e.g., inheritance, inner classes, conditional statements,

[17] Since some of the samples in the AMD dataset predate the introduction of the ART system, it is possible that this app uses an obfuscation method that is only compatible with older versions of Android, and that it would fail the more strict verification performed by the ART compiler.

etc.). The strong relationship between decompilation failure rate and method size, shown in Fig. 4, further suggest that resource exhaustion is a major cause of decompilation failures. In our follow-up study on reasons for decompilation failures, we wanted to quantify the degree to which resource exhaustion and other implementation-level deficiencies contributed to decompilation failures. The results of our analysis is presented here.

### 5.1 Classification of Exceptions

We collected all exception types mentioned in error messages, as described in Section 3.2. Table 13 shows the frequencies of each exception type, for each of the decompilers.

The table also shows which exception types we have classified as "anticipated". While this classification is necessarily somewhat subjective, we consider exceptions accompanied by a meaningful error message, describing the reason why decompilation was aborted for the particular method, as anticipated. Error messages not mentioning an exception are also considered as anticipated. On the other hand, exceptions accompanied by either no message, or a non-meaningful message[18], were considered as unexpected.

The rationale for our classification approach is that we found that each decompiler consistently used a specific (small) set of exception types (either built-in or custom) for reporting "gracefully" handled errors, while the remaining exception types appeared to be the result of some unhandled corner case, e.g., passing invalid or illegal parameters to the interface of some abstract data type. There also appeared to be little overlap between the two classes, i.e., a given exception type tended to either always be accompanied by a meaningful error message, or never be.

Turning our attention again to Table 13, we see that CFR is the only decompiler that predominantly excludes the exception name from error messages. Moreover, it is the decompiler with the highest number of errors classified as anticipated (over 99%). In contrast, out of the errors reported by Fernflower, about 29% are classified as unexpected. It should be noted, however, that Fernflower does not provide any details of errors in its error messages. Instead, it simply emits a `RuntimeException` with the message "parsing failure!", making classification hard. Most of Fernflower's unexpected errors are due to null pointer exceptions.

The best-performing decompiler Jadx had the greatest percentage of unexpected errors, around 80%, the vast majority of which are null pointer exceptions. Jadx uses several custom exception types for reporting errors, and also shows the greatest diversity in the exceptions it throws.

Finally, Procyon, which mostly uses the built-in `IllegalStateException` to report anticipated errors, had around 35% unexpected failures.

### 5.2 Differences in Co-Failure Rates

Table 14 breaks down co-failures on the AndroZoo dataset into anticipated and unexpected failures. The top of the table shows, for each decompiler, the percentage of anticipated failures that coincided with a failure (of either class) in the other decompilers. The bottom

[18]For example, the following error message was frequently emitted by Jadx: "NullPointerException in pass: SimplifyVisitor, details: java.lang.NullPointerException: null".

**Table 13** Exception counts for the different decompilers on the AndroZoo dataset. (Custom exception types are highlighted in italics)

| Decompiler | Exception | Count | Anticipated |
|---|---|---|---|
| CFR | ArrayIndexOutOfBoundsException | 8,986 | |
| | ClassCastException | 3,604 | |
| | *ConfusedCFRException* | 582,274 | ✓ |
| | IllegalStateException | 8,764 | ✓ |
| | IndexOutOfBoundsException | 211 | |
| | NullPointerException | 12 | |
| | NumberFormatException | 1 | |
| | UnsupportedOperationException | 1,365 | |
| | **MESSAGE** | 2,655,306 | ✓ |
| Fernflower | ArrayIndexOutOfBoundsException | 85 | |
| | IndexOutOfBoundsException | 2 | |
| | NullPointerException | 525,298 | |
| | OutOfMemoryError | 1,101 | |
| | RuntimeException | 1,295,330 | ✓ |
| Jadx | ArrayIndexOutOfBoundsException | 3,614 | |
| | BufferUnderflowException | 15 | |
| | *CodegenException* | 1,655 | ✓ |
| | *DecodeException* | 8 | ✓ |
| | IllegalArgumentException | 1,133 | ✓ |
| | IndexOutOfBoundsException | 11 | |
| | *JadxOverflowException* | 3,371 | ✓ |
| | *JadxRuntimeException* | 11,845 | ✓ |
| | NegativeArraySizeException | 22 | |
| | NullPointerException | 68,608 | |
| | StackOverflowError | 2,129 | |
| | StringIndexOutOfBoundsException | 4 | |
| | **MESSAGE** | 582 | ✓ |
| Procyon | *AdaptFailure* | 463 | |
| | ArrayIndexOutOfBoundsException | 355 | |
| | ConcurrentModificationException | 21,486 | |
| | IllegalArgumentException | 1 | ✓ |
| | IllegalStateException | 1,234,464 | ✓ |
| | IndexOutOfBoundsException | 583,076 | |
| | NullPointerException | 56,617 | |
| | NumberFormatException | 241 | |
| | StackOverflowError | 178 | |
| | UnsupportedOperationException | 15,330 | |

**Table 14** Co-failure rates of anticipated and unexpected failures on the AndroZoo dataset

| | CFR | Fernflower | Jadx | Procyon | |
|---|---|---|---|---|---|
| CFR | 100 | 22.31 | 0.23 | 39.19 | Anticipated |
| Fernflower | 47.67 | 100 | 0.17 | 44.77 | |
| Jadx | 26.38 | 3.02 | 100 | 7.35 | |
| Procyon | 60.93 | 31.07 | 0.11 | 100 | |
| CFR | 100 | 10.64 | 0.13 | 20.42 | Unexpected |
| Fernflower | 20.57 | 100 | 0.10 | 20.16 | |
| Jadx | 3.40 | 2.87 | 100 | 2.43 | |
| Procyon | 77.18 | 44.63 | 0.27 | 100 | |

part of the table shows corresponding figures for unexpected failures. We do not consider timeouts as failures in this analysis.

Comparing with Table 9 (Excluding timeouts), we see that for all decompilers except Procyon, the co-failure rates are higher for anticipated errors, compared to the overall co-failure rate, and vice versa for unexpected errors. For Procyon, the opposite is true, although the relative differences compared to the overall co-failure rate are not that big for either class of failures.

Based on these figures, it would appear that a major part of the diversity between decompilers discussed in Section 4.3 is due to unexpected errors. That is, the predominant reason why an ensemble improves the decompilation success rate is due to various corner-cases, which might result in unexpected failures in one decompiler, but be successfully handled by other decompilers. On the flip-side, decompilers differing in their view of what constitutes “impossible-to-decompile” code, appears to be a less prominent reason for decompiler diversity.

## 5.3 Resource-Exhaustion Failures

To quantify the frequency of the aforementioned resource-exhaustion problems, we gathered and manually checked each unique error message containing any of the words “heap”, “memory”, “overflow”, “limit”, or “recursion”, looking for errors such as reaching a maximal recursion-depth, stack overflows, or out-of-memory errors. CFR had no such errors, while Fernflower had 1,342 (0.07% of all decompilation failures). Jadx had the proportionally greatest number of resource-exhaustion problems: 5,762 or 6.2%. Finally, Procyon had only 294 such errors, or 0.02% of all failures.

While Jadx exhibited a fair number of resource-exhaustion problems on the AndroZoo malware apps, this turned out not to be the major cause of failures. The main cause instead appeared to be other types of unhandled corner-cases, leading to unanticipated erros. Null-pointer exceptions, for example, comprised more than 70% of all Jadx’s decompilation failures. The other decompilers exhibited very few or no resource-exhaustion errors. It should be noted, however, that both Fernflower and Procyon had a large number of timeouts on several datasets. This could also be indicative of some kind of resource-exhaustion problem in those decompilers

## 6 Summary and Discussion

In this section, we first summarize our findings. We then discuss some threats to validity, and finally outline some directions for future work.

### 6.1 Summary of Results

Here, we summarize the main findings of our work, in the context of our research questions.

**RQ1:** ***To what degree can we expect decompilers to successfully recover source code from Android apps?***

The native Android decompiler Jadx performed very well in our study with a (weighted) average of 0.04% failed methods per app, while the Java decompilers had mean failure rates of around 1%. The failure rates varied substantially between our three datasets, however, with Jadx having mean failure rates that, compared to the open source apps, were around 2x and 20x, for Google Play and malware apps, respectively. Moreover, Jadx could successfully decompile every method (as reported by apkanalyzer) in around 75% of the open-source apps. Interestingly, around 80% of the malware apps could also be fully decompiled. However, for the Google Play apps, which tended to be larger and have more methods, only about one app in five could be fully decompiled by Jadx.

**RQ2:** ***To what degree is decompilation-breaking obfuscation a concern when analyzing malware or commercial apps for the Android platform?***

Our manual analysis revealed several cases of code that could not be decompiled because it was obfuscated. Moreover, the increased failure rates of commercial apps, and even higher failure rates of malware, which could not be explained by other factors, would indicate that obfuscation is a factor. Likewise, the higher failure rates of ad-supported apps, whose developers would have a stronger incentive to protect their code from, e.g., ad-fraud, also points towards obfuscation being a factor. A similar increase in incidence for packed apps could also be seen, especially for the new AndroZoo dataset. As commercial packers are known to make heavy use of obfuscation (Duan et al. 2018; Yang et al. 2015; Zhang et al. 2015), this is another sign pointing towards obfuscation being a factor.

During the manual analysis performed in the original study, we found that most failures due to obfuscation appeared to be caused by the same decompiler limitations that cause failures on unmodified code, rather than by a deliberate attempt to prevent decompilation. This conclusion was also corroborated by the fact that, in several cases where a decompiler failed due to obfuscation, at least one other decompiler succeeded on the same code. In particular, we discovered no cases of advanced control-flow obfuscation using "fake" branches to invalid code locations, which are commonly encountered in obfuscated native or JVM code.

Our follow-up study on the set of malicious apps from AndroZoo, on the other hand, indicated more strongly that code obfuscation might be a problem when analyzing Android malware. This dataset had an overall significantly higher failure rate than all the other datasets. Furthermore, in our analysis of tokens associated with failures (Section 4.5), the other three datasets displayed a pattern of failures indicating that encoding or parsing code was a major contributor to failures. The fact that this pattern was not observed for the AndroZoo apps makes obfuscation more likely as a culprit. It should be noted, however, that despite the stronger evidence of obfuscation in this dataset, 96% of all apps in the set could still be fully decompiled by an ensemble of decompilers. Therefore, our conclusion is that, decompilation-breaking obfuscation is a potential concern that must be taken into account when analyzing Android malware. However, for the vast majority of malware samples, complete decompilation of all methods can still be achieved, at least when using an ensemble of decompilers.

It should be noted, however, that it was much more common for apkanalyzer to fail on the malware than on the benign apps (in around 5% and 2% of cases, respectively, for the AMD and AndroZoo sets). This kind of failure is a strong indicator of obfuscated or otherwise manipulated APKs. Likewise, while packing was rarely encountered for the other datasets, around 3.5% of AndroZoo malware matched a packer signature. As packing completely prevents static analysis of an app's code, it is obviously a strong deterrent against decompilation.

Finally, it should also be noted that successful decompilation of a method does not necessarily imply that the result is *useful* for subsequent manual or automated analyses. Advanced obfuscation techniques, such as the one suggested by Balachandran et al. (2016), which route control flow through a large number of `try-catch` blocks, can effectively hide a method's static control flow, even if the source code can be completely recovered by decompilation.

**RQ3:** ***Do different Android decompilers tend to systematically fail on the same methods, or do their results complement each other?***

In our experiments, an ensemble of decompilers was able to improve the decompilation success rate, so that between 85% and 98% of all apps could be fully decompiled, depending on the dataset. Even though Jadx outperformed the other decompilers by a broad margin, our results showed that in 96% of cases where Jadx failed to decompile a method, at least one of the other decompilers succeeded. Our analysis of co-failure rates indicated that, in many cases, code that induced excessive computation times in some of the decompilers would instead be detected as erroneous, and lead to a reported failure, in some of the other decompilers. Our follow-up study also indicated that the predominant reason for decompiler diversity is that decompilers differ in their ability to handle various corner cases that might cause unexpected failures in some decompilers, but be successfully handled by others.

**RQ4:** ***To what degree does implementation-level limitations, in contrast to fundamental algorithmic limitations, contribute to decompilation failures?***

Our follow-up study on the AndroZoo dataset revealed that, among the four decompilers, a varying degree of failures appeared to be caused by unhandled exceptions (i.e., bugs). CFR had only a very small amount of such failures, while Fernflower and Procyon both had around 30% unexpected failures. For the best-performing decompiler Jadx, 80% of failures appeared to be due to unexpected exceptions being thrown.

While our original study indicated that various resource-exhaustion problems could be a common cause of failures, our follow-up study revealed only a small amount of such failures. However, given the large number of timeouts exhibited by some decompilers, it is possible that our 5-minute timeout limit might have masked some resource-exhaustion bugs.

### 6.2 Threats to Validity

The limitations of our methodology, which we have already discussed in Section 3.1.3, pose a threat to the internal validity of our results. However, we believe that the imprecision introduced by these shortcomings does not invalidate the main conclusions of our work.

A potential threat to the external validity of our original study was the representativeness of datasets, where our main concern was that the AMD dataset was a few years old, and might no longer fully reflect, for example, obfuscation techniques used in present-day malware. We have sought to remedy that problem in this work by including a more recent malware dataset. However, even with a recent collection of malicious apps, it is difficult to know the degree to which the dataset is a representative subset of current in-the-wild malware.

Finally, we observed in our study that code in third-party libraries was a major contributor to decompilation failures. As libraries are often of less interest when using a decompiler to analyze an app, including them in the analysis might make the results less representative of decompiler performance in practice. Moreover, the same library could contribute to decompilation failures in several apps.

### 6.3 Future Work

One direction of future work would be to improve the matching accuracy of our approach by not relying on textual matching of method signatures. Since DEX files contain unique identifiers for each method, which a decompiler must access at some point, we could use this to achieve better unification of results. As this would require an in-depth understanding of the code-base for all studied decompilers, and likely also non-trivial modifications to their source code, we left it for future work in this study.

In this work, we only considered whether or not a decompiler *reported* a method as unsuccessfully decompiled. Another topic of interest for future work is to assess the correctness of recovered source code, as has already been done by others (Hamilton and Danicic 2009; Kostelanský and Dedera 2017; Harrand et al. 2019) for JVM bytecode decompilation.

As noted in the preceding section, third-party libraries could introduce imprecision in the results of our study. Therefore, one direction for future work could be to integrate existing techniques (Backes et al. 2016; Li et al. 2017) for detecting third-party libraries into our analysis platform.

Finally, as we discuss in the introduction (Section 1), apps can also contain native code components. Putting functionality in native-code libraries, potentially in combination with obfuscation, is an effective way to hinder decompilation. In particular, many commercial packers are known to implement their unpacking logic in obfuscated native code Xue et al. (2017). As the use of native-code components is widespread in modern Android apps (for example, around 60% of the apps in the Google Play dataset and around 45% of the AndroZoo malware contain native-code components), studying the effectiveness of decompilers or disassemblers[19] on native app code would be an interesting avenue of future work.

## 7 Related Work

An early study of Java decompilation correctness was performed by Hamilton and Danicic (2009). Kostelanský and Dedera (2017) performed a similar study, and concluded that the correctness of state-of-the-art decompilers had improved significantly between 2009 and 2017. Naeem et al. (2007) proposed several metrics for measuring decompiler performance. Gusarovs (2018) compared the success rate and correctness of four Java decompilers on a number of manually-crafted test cases. Harrand et al. (2019) performed a large-scale study of 8 Java decompilers, in which they assessed both the syntactic and semantic correctness of recovered source code. Their study revealed that the best decompiler (CFR) could only

[19] Similar to, for example, the works of Andriesse et al. (2016) and Pang et al. (2021) for x86 Linux and Windows executables, and Jiang et al. (2020) for ARM binaries (including Android system libraries).

produce syntactically correct code for 84% of classes. Similar to our results for decompilation success-rate, however, the study also showed that this figure could be significantly improved by combining the output from several decompilers.

Jang et al. (2019) recognized that popular Android decompilers failed to decompile a significant portion of methods in many apps, and proposed the Kerberoid system, which uses an ensemble of three Android decompilers to improve decompilation success rate. While their method was only evaluated on 151 open-source apps, our large-scale study on a wider range of apps confirmed their finding that an ensemble of decompilers can often improve the success rate significantly. A notable difference between our results and theirs, however, is that we found failures to be much more rare. (They reported, for example, that Jadx only managed to recover half of the methods for 10% of the apps, and that it could recover all methods of an app in only 8% of cases.) We suspect that this might be due to differences in the way success rate was measured. While we recovered the signatures of failed methods from error messages and matched those against the method signatures reported by apkanalyzer, Jang et al. appears to have used the opposite approach of scanning for successfully decompiled methods in recovered source code (which we opted against, due to the many corner cases and potential failure modes, as discussed in Section 3.1.3).

A more advanced version of the decompiler-ensemble concept was proposed by Harrand et al. (2020) in a follow-up to their previous study. They introduce *meta-decompilation* as way to merge the results from several Java decompilers, in order to improve overall decompiler correctness.

Dong et al. (2018) performed a large scale study of the prevalence of Android obfuscation. While we were mainly concerned with anti-decompilation obfuscation in this work, they instead focused on identifier renaming, string encryption, Java reflection, and packing. Out of those obfuscation techniques, only string encryption stood out as significantly more common in malware.

Finally, a recent study by Hammad et al. (2018) showed that applying advanced obfuscation techniques, such as control-flow obfuscation, frequently tended to break apps, so that they would fail to install or run. This is in line with our findings, which indicate that such techniques are rarely used in the wild.

## 8 Conclusion

In this work we have presented the results of a large-scale study of the decompilation success rate of four different compilers on four large sets of Android apps. While the state-of-the-art Android decompiler Jadx achieved a very low failure rate of only 0.04% failed methods on average, it still failed to fully decompile many apps. We also corroborated earlier results, which indicated that decompilers exhibit a great deal of diversity in the apps and methods that they fail on. Our follow-up study revealed that the dominant reason for this diversity appear to be decompiler bugs, which cause unexpected failures due to unforeseen corner cases, and that decompilers differ in terms of which corner cases they can handle. Finally, our empirical results and complementary manual investigation indicate that deliberate anti-decompilation obfuscation is not a major cause of decompilation failures in commercial apps. Instead, it appears that most failures on such apps happen because current decompilers have technical limitations that sometimes prevent them from successfully processing methods that are large, have complex control flow, or exhibit deep levels of various kinds of nesting. For malicious apps, however, code obfuscation might be more of a concern, even though the vast majority of malware apps in our study could still be fully decompiled.

**Acknowledgements** This work was partially supported by the Swedish Graduate School in Computer Science (CUGS) and the Excellence Center at Linköping – Lund in Information Technology (ELLIIT).

**Funding** Open access funding provided by Linköping University

## Declarations

**Conflict of Interests** The authors declare that they have no conflict of interest.

## References

Allix K, Bissyandé TF, Klein J, Le Traon Y (2016) Androzoo: collecting millions of android apps for the research community. In: Proceedings of the 13th international conference on mining software repositories, ACM, pp 468–471

Andriesse D, Chen X, Van Der Veen V, Slowinska A, Bos H (2016) An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: 25th USENIX security symposium (USENIX security 16), pp 583-600

Backes M, Bugiel S, Derr E (2016) Reliable third-party library detection in Android and its security applications. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp 356–367

Balachandran V, Tan DJ, Thing VL et al (2016) Control flow obfuscation for android applications. Comput Security 61:72–93

Cen L, Gates CS, Si L, Li N (2015) A probabilistic discriminative model for android malware detection with decompiled source code. IEEE Trans Dependable Secure Comput 12(4):400–412

Chan JT, Yang W (2004) Advanced obfuscation techniques for java bytecode. J Syst Softw 71(1-2):1–10

Chen S, Fan L, Chen C, Su T, Li W, Liu Y, Xu L (2019) StoryDroid: automated generation of storyboard for android apps. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE), pp 596–607

Collberg C, Thomborson C, Low D (1997) A Taxonomy of Obfuscating Transformations. Tech rep Department of Computer Science. The University of Auckland, New Zealand

Dong S, Li M, Diao W, Liu X, Liu J, Li Z, Xu F, Chen K, Wang X, Zhang K (2018) Understanding android obfuscation techniques: a large-scale investigation in the wild. In: Security and privacy in communication networks. Springer international publishing, pp 172–192

Duan Y, Zhang M, Bhaskar AV, Yin H, Pan X, Li T, Wang X, Wang X (2018) Things you may not know about android (un) packers: a systematic study based on whole-system emulation. In: Network and distributed system security symposium

Enck W, Octeau D, McDaniel P, Chaudhuri S (2011) A study of android application security. In: USENIX security symposium

Gamba J, Rashed M, Razaghpanah A, Tapiador J, Vallina-Rodriguez N (2020) An analysis of pre-installed android software. In: 2020 IEEE symposium on security and privacy (SP), pp 1039–1055

Gibler C, Crussell J, Erickson J, Chen H (2012) AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In: Trust and trustworthy computing, Springer Berlin, Heidelberg, Berlin. pp 291–307

Gusarovs K (2018) An analysis on java programming language decompiler capabilities. Appl Comput Syst 23(2):109–117

Hamilton J, Danicic S (2009) An evaluation of current java bytecode decompilers. In: 2009 Ninth IEEE international working conference on source code analysis and manipulation, pp 129–136

Hammad M, Garcia J, Malek S (2018) A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In: Proceedings of the 40th international conference on software engineering, pp 421–431

Harrand N, Soto-Valero C, Monperrus M, Baudry B (2019) The strengths and behavioral quirks of Java bytecode decompilers. In: 2019 19th International working conference on source code analysis and manipulation (SCAM), pp 92–102

Harrand N, Soto-Valero C, Monperrus M, Baudry B (2020) Java decompiler diversity and its application to meta-decompilation. J Syst Softw 168:110645

Hou TW, Chen HY, Tsai MH (2006) Three control flow obfuscation methods for java software. IEE Proc-Softw 153(2):80–86

Jang H, Jin B, Hyun S, Kim H (2019) Kerberoid: a practical android app decompilation system with multiple decompilers. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security, pp 2557–2559

Jiang M, Zhou Y, Luo X, Wang R, Liu Y, Ren K (2020) An empirical study on arm disassembly tools. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp 401–414

Junod P, Rinaldini J, Wehrli J, Michielin J (2015) Obfuscator-LLVM–software protection for the masses. In: 2015 IEEE/ACM 1st international workshop on software protection, IEEE, pp 3–9

Kostelanský J, Dedera Ĺ (2017) An evaluation of output from current java bytecode decompilers: is it android which is responsible for such quality boost? In: 2017 Communication and information technologies (KIT), pp 1–6

Li M, Wang W, Wang P, Wang S, Wu D, Liu J, Xue R, Huo W (2017) LibD: scalable and precise third-party library detection in android markets. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE), pp 335–346

Linn C, Debray S (2003) Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM conference on computer and communications security, pp 290–299

Martín A, Menéndez HD, Camacho D (2017) MOCDRoid: multi-objective evolutionary classifier for android malware detection. Soft Comput 21(24):7405–7415

Mauthe N, Kargén U, Shahmehri N (2021) A large-scale empirical study of android app decompilation. In: 2021 IEEE international conference on software analysis, evolution and reengineering (SANER), pp 400–410

Ming J, Xu D, Wang L, Wu D (2015) Loop: logic-oriented opaque predicate detection in obfuscated binary code. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security, pp 757–768

Naeem NA, Batchelder M, Hendren L (2007) Metrics for measuring the effectiveness of decompilers and obfuscators. In: 15th IEEE international conference on program comprehension (ICPC ’07), pp 253–258

Pang C, Yu R, Chen Y, Koskinen E, Portokalidis G, Mao B, Xu J (2021) SoK: all you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In: 2021 IEEE symposium on security and privacy (SP), pp 833–851

Pauck F, Bodden E, Wehrheim H (2018) Do android taint analysis tools keep their promises? In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 331–341

Roundy KA, Miller BP (2013) Binary-code obfuscations in prevalent packer tools. ACM Comput Surveys (CSUR) 46(1):1–32

Shan Z, Neamtiu I, Samuel R (2018) Self-hiding behavior in android apps: detection and characterization. In: Proceedings of the 40th international conference on software engineering, pp 728–739

Tian DJ, Hernandez G, Choi JI, Frost V, Raules C, Traynor P, Vijayakumar H, Harrison L, Rahmati A, Grace M et al (2018) ATTention spanned: comprehensive vulnerability analysis of AT commands within the android ecosystem. In: 27th USENIX security symposium (USENIX security 18), pp 273–290

Wang H, Guo Y, Ma Z, Chen X (2015) WuKong: a scalable and accurate two-phase approach to android app clone detection. In: Proceedings of the 2015 international symposium on software testing and analysis, association for computing machinery, ISSTA 2015, pp 71–82

Wei F, Li Y, Roy S, Ou X, Zhou W (2017) Deep ground truth analysis of current Android malware. In: Detection of intrusions and malware, and vulnerability assessment, Springer international publishing, pp 252–276

Xue L, Luo X, Yu L, Wang S, Wu D (2017) Adaptive unpacking of android apps. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE), pp 358–369

Yang W, Zhang Y, Li J, Shu J, Li B, Hu W, Gu D (2015) Appspear: bytecode decrypting and dex reassembling for packed android malware. In: Research in attacks, Intrusions, and Defenses, Springer international publishing, p 359–381

Zhang Y, Luo X, Yin H (2015) Dexhunter: toward extracting hidden code from packed android applications. In: Computer security – ESORICS 2015, Springer international publishing, pp 293–311