A Large-Scale Empirical Study of Android App Decompilation

1st Noah Mauthe
Saarland University
Saarbrücken, Germany
s8nomaut@stud.uni-saarland.de

2nd Ulf Kargén
Linköping University
Linköping, Sweden
ulf.kargen@liu.se

3rd Nahid Shahmehri
Linköping University
Linköping, Sweden
nahid.shahmehri@liu.se

***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. While a large proportion of methods in an app can typically be decompiled successfully, it is common that at least some methods fail to decompile. In order to better understand the practical applicability of techniques in which decompilation is used as part of an automated analysis, it is important to know the actual expected failure rate of Android decompilation. To this end, we have performed what is, to the best of our knowledge, the first large-scale study of Android decompilation failure rates. We have used three sets of apps, consisting of, respectively, 3,018 open-source apps, 13,601 apps from a recent crawl of Google Play, and a collection of 24,553 malware samples. In addition to the state-of-the-art Dalvik bytecode decompiler jadx, we used three popular Java decompilers. While jadx achieves an impressively low failure rate of only 0.02% failed methods per app on average, we found that it manages to recover source code for all methods in only 21% of the Google Play apps.**

We have also sought to better understand the degree to which in-the-wild obfuscation techniques can prevent decompilation. Our empirical evaluation, complemented with an in-depth manual analysis of a number of apps, indicate that code obfuscation is quite rarely encountered, even in malicious apps. Moreover, decompilation failures mostly appear to be caused by technical limitations in decompilers, rather than by deliberate attempts to thwart source-code recovery by obfuscation. This is an encouraging finding, as it indicates that near-perfect Android decompilation is, at least in theory, achievable, with implementation-level improvements to decompilation tools.

***Index Terms*—Android, mobile apps, decompilation, obfuscation, reverse engineering, malware**

I. Introduction

Decompilers, i.e., tools that can reconstruct the source code from a program binary, are ubiquitous aids in malware analysis and app security auditing for the Android platform. Android decompilers are also used extensively in academia to lift the *Dalvik* bytecode of Android apps into Java source code, prior to manual or automated inspection [1]–[7]. In the literature, decompilation of Android apps is frequently considered a "solved" problem, in that it is simply expected that source code can be accurately recovered from an app. While Android decompilers are generally known to perform well, it is also known that their success rate is often not 100%, especially for obfuscated or heavily optimized apps.

In many works [2], [8]–[12], decompilation is used as the first step in an automated analysis pipeline. In order to understand the failure modes of such approaches, and to allow better estimates of their efficacy in the general case, a fundamental question is: *To what degree can we expect decompilers to successfully recover source code from Android apps?* To the best of our knowledge, there have been no previous systematic attempts at answering this question. In this paper, we strive towards filling this knowledge gap by presenting the first large-scale study of Android app decompilation success rate, where the above question serves as our primary research question (**RQ1**).

On the PC platform, control-flow obfuscation is frequently used by both malware authors and legitimate software developers to prevent decompilation or disassembly of native code [13], [14]. Similarly, several obfuscation techniques exist for the Java virtual machine [15], [16], which are able to prevent decompilation of Java bytecode back into legible source code. Therefore, obfuscation is a major hurdle when attempting decompilation (or related techniques, such as control-flow graph reconstruction), on PC malware or DRM-protected commercial software.

Control-flow obfuscation is technically more challenging, albeit not impossible, to implement for Dalvik bytecode due to the so-called *register-type conflict* problem [17] (which we briefly describe in Section II). For this reason, the most commonly used obfuscation techniques for Android apps, such as identifier renaming or string encryption, aim primarily to hide clues about program semantics from human analysts, rather than preventing decompilation per se. The question remains, however: *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**).

Our third and final research question concerns the performance of individual decompilers. It has been shown that various idiosyncrasies of Java decompilers can cause significant differences in relative performance between decompilers, depending on the program being analyzed [18]. Moreover, Jang et al. [19] showed in a study on 151 open-source apps that an ensemble of decompilers outperformed each individual de-

compiler. To determine if the results of their small-scale study 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**)

In summary, the main contributions of this paper are as follows:

- We perform a large-scale study of the decompilation success rate for Android apps using four different decompilers. Our evaluation is 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.
- We characterize the differences in decompilation success rate between the three datasets, and perform a preliminary analysis of potential causes of these differences.
- We complement our statistical analysis with a manual analysis of a number of Android apps.
- Finally, we make our implementation and collected data available in the interest of open science[1].

II. 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 Dalvik Executable (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 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, Java decompilers can often also be used on Android apps by first converting the Dalvik bytecode into equivalent bytecode for the Java virtual machine (JVM), using a tool such as ded [12] or dex2jar [20]. 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 an entire DEX file is stored in encrypted form, which is 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 [21]. 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 [22]. On Android, however, this technique is considerably harder to implement due to the aforementioned 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 exist [17], it is unclear to what degree, if any, this type of anti-decompilation technique is used in the wild for Android apps.

III. Methodology

In this section, we outline the methodology used in our study. We begin with a detailed description of our approach, and then we discuss some of its limitations.

A. Approach

As depicted in Figure 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 [23] and 13,601 apps from the Google Play store. Finally, we used an existing dataset [24] of Android malware, consisting of 24,553 samples collected between 2010 and 2016.

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. In order to get a comprehensive overview of the most popular applications

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

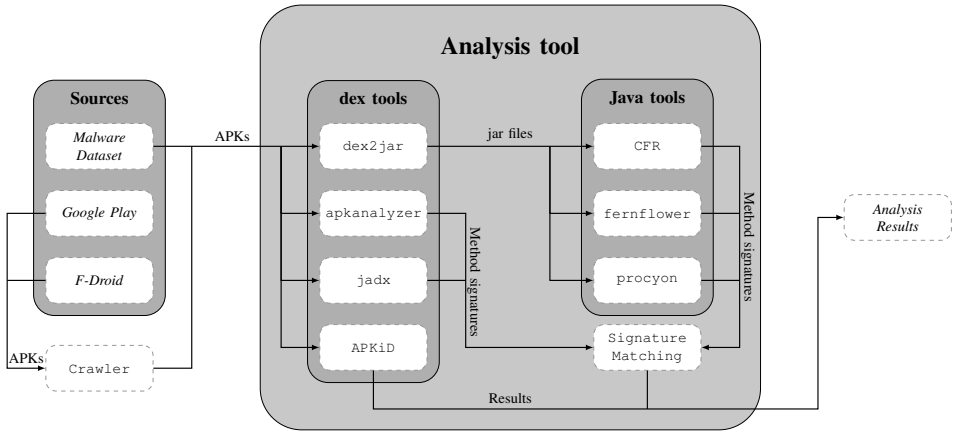

Fig. 1. An overview of our analysis approach.

in the store, we used an approach similar to, e.g., Backes et al. [25] and crawled Google Play by category. Our crawler first retrieves the current set of 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[2], 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.

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.

In the next step, each app is decompiled with four different decompilers. In addition to the state-of-the art native Android decompiler jadx [27], we also used the three popular Java decompilers CFR [28], Fernflower [29] and Procyon [30]. 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 [31], 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 use apkanalyzer [32] from the Android SDK to extract signatures for every method in an app. 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.

B. 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. It should be noted that matching the successfully decompiled methods in addition to the failed ones would be quite challenging, as this 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 be fully compliant Java code.

[2]We deemed this a reasonable restriction, considering that only 3.9% of all Google Play apps are paid [26].

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 V.

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.

IV. Results

In this section, we present the results of our empirical study.

A. Basic Dataset Statistics

Table I shows some basic properties of our three 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 malware 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.

TABLE I
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
malware	24,553	1,220	33	23,333	131

TABLE II
TIMEOUT STATISTICS FOR THE 4 DECOMPILERS

Dataset	CFR	Fernflower	Jadx	Procyon
f-droid	1	164	0	37
google	21	7652	4	1419
malware	8	1955	1	130

The number of timeouts for each dataset and decompiler are shown in Table II. The native Dalvik decompiler jadx performs the best with only 5 timeouts. CFR also performs well with only a few timed-out apps. Fernflower, on the other hand, experiences a very large 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 III-B are broken down in Tables III and IV. 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.

TABLE III
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	15.746
malware	0.0	16.905	0.0	22.154

TABLE IV
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	6.609	0.331	34.030	9.060
malware	1.586	0.036	2.649	1.670

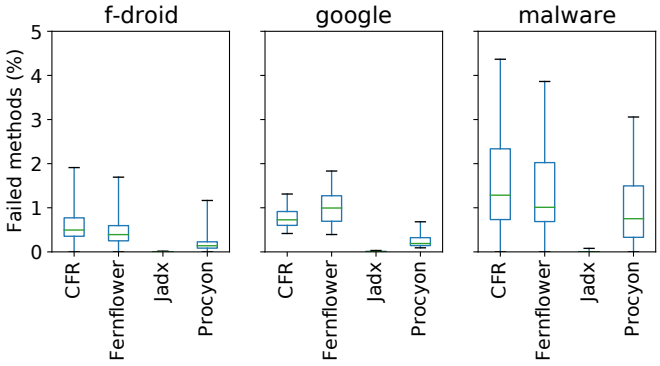

Fig. 2. Failure rate distributions for the 4 decompilers, excluding timeouts and dex2jar failures. Whiskers show the upper and lower 5th percentiles.

TABLE V
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
malware	1.751	1.459	0.047	1.078
weighted avg.	1.094	1.024	0.021	0.564

B. 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 V 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 0.02% 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 dataset. We elaborate on this further in Section IV-D.

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

C. 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 VII 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

TABLE VI
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
malware	1.838	9.726	0.044	1.672
weighted avg.	1.204	24.703	0.023	4.733

TABLE VII
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	74.69	8.65
google	21.02	21.03	0.15
malware	78.81	79.68	0.60

TABLE VIII
DECOMPILER CO-FAILURE PERCENTAGE 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

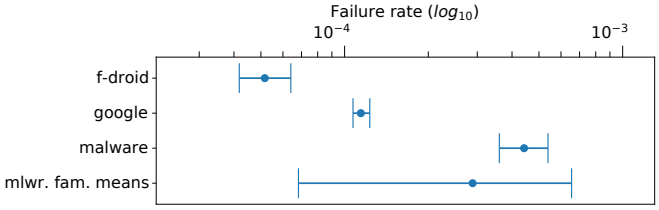

Fig. 3. 95% confidence intervals for jadx mean failure rates.

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, almost 80% of malware apps could also be fully decompiled by jadx. The fact that the malware apps had significantly fewer methods on average (5,142), 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% of the entire malware dataset.)

The next column in Table VII shows the number of apps that could be fully decompiled by combining the results from all decompilers. As is evident from the table, this only negligibly improves the success rate. The last column shows the percentage of apps that could be fully decompiled by all decompilers. These figures are negligible, except for the F-Droid dataset.

Another way to characterize the diversity of decompilers is to measure their co-failure rate. 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 III-B. Also, 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.

Table VIII shows, for each decompiler, 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 (c.f. Table VI).

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.

D. 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 V. We note that the mean failure rate of Google Play apps is roughly twice that of the open-source apps, while for the malware apps, the mean failure rate is roughly one order of magnitude higher.

As seen above in Figure 2, the failure rates vary significantly between different apps. Therefore, sampling error might be a concern when characterizing differences between the datasets. In order to quantify the effect of sampling error, we have computed 95% confidence intervals (shown as error bars in Figure 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 three datasets.

Another potential concern is that the distribution of samples over different malware families is highly skewed in our set of malicious apps. For example, around one third of the malware apps belong to the same family. Therefore, we have also included a weighted mean, which is computed by taking the mean of the family-wise mean failure rate. The result is shown in the lowermost bar of Figure 3. 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. For malware apps, 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, 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.

E. 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 Figure 4 shows the results. A strong, roughly linear dependence between method size and failure rate is evident in the log-log scale bar chart. The 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 Figure 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 malware dataset are about one order of magnitude higher compared to the other datasets, which corresponds well with the results shown in Figure 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 "." (dot) 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. The top 20 tokens for each dataset are shown in Table IX.

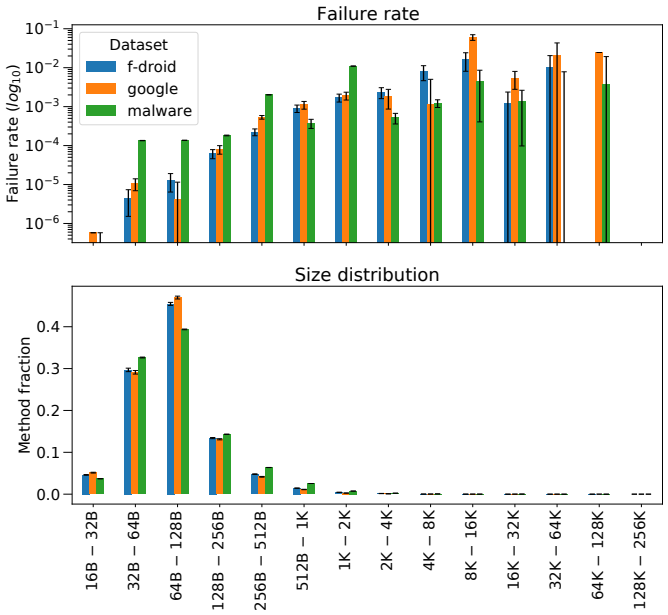

Fig. 4. jadx failure rate as a function of binned method sizes (top), and the distribution of method sizes, using the same bins (bottom).

Several interesting patterns can be identified. We see that the tokens `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 malware 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 suggest 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 V). The 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 malware 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.

V. Manual Analysis

In this section we describe our complementary manual analysis.

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 malware 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 [33]. 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 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.

Malware. 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 III-B. 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[3].

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).

VI. 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.

[3]Since some of the samples in the malware 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.

TABLE IX
TOP 20 CLASS/PACKAGE IDENTIFIER TOKENS ASSOCIATED WITH JADX DECOMPILATION FAILURES.

f-droid			google			malware		
Token	Failures	Frequency	Token	Failures	Frequency	Token	Failures	Frequency
SlidingWindowKt	183	9.30%	apptimize	2,595	1.72%	BugSenseHandler	2,034	1.50%
windowedIterator	183	20.29%	SlidingWindowKt	1,901	9.14%	Encoder	544	2.33%
ReaderBasedJsonParser	83	1.56%	windowedIterator	1,863	19.49%	ZLDTDParser	438	50.00%
NonBlockingJsonParser	48	3.52%	ReaderBasedJsonParser	1,390	1.60%	ReaderBasedParser	426	3.30%
MergerBiFunction	37	27.41%	NonBlockingJsonParser	1,112	3.48%	jianmo	283	1.07%
ASN1Set	28	1.48%	BaseListBitmapDataSubscriber	849	30.53%	igexin	251	1.24%
ConstructedOctetStream	28	12.17%	zzdfh	772	5.54%	Utf8StreamParser	216	1.02%
ConverterSet	28	6.22%	MergerBiFunction	664	22.13%	imobile	142	1.75%
fixedPeriodTicker	25	7.91%	ConverterSet	553	8.42%	base64	107	1.63%
FlowKt__DelayKt	25	1.24%	zzdph	485	2.24%	Provider	98	2.38%
Fx	23	3.01%	zzdbm	446	2.69%	WbxmlParser	60	1.15%
InterruptibleTask	22	4.12%	zzdme	422	3.67%	SDK	52	2.10%
LDAPStoreHelper	19	1.44%	zzdha	332	1.05%	products	52	10.18%
X509LDAPCertStoreSpi	19	3.63%	JSONLexerBase	290	3.37%	threegvision	52	10.18%
BaseListBitmapDataSubscriber	18	31.58%	InterruptibleTask	283	3.09%	inigma_sdk	52	10.18%
AbstractListeningExecutorService	14	1.88%	ASN1Set	241	2.31%	rc	49	3.38%
DSAParametersGenerator	12	1.46%	MethodWriter	236	1.31%	QueueDetails	49	7.40%
TokenStream	11	1.14%	JSONLexer	205	2.52%	QueueOverview	49	5.94%
ASN1Object	10	1.69%	fixedPeriodTicker	198	6.32%	FaultTolerantNegotiator	48	7.10%
NioClientManager	10	7.69%	zzflf	160	38.37%	qqmagic	42	3.20%

A. 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 0.02% 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 10x, 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, almost 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.

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.

However, both our statistical and manual analyses indicate that most decompilation failures are 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, etc.). The strong relationship between decompilation failure rate and method size, shown in Figure 4, further suggests that resource exhaustion is a major cause of decompilation failures.

Moreover, in all cases, except for the one heavily obfuscated malware sample that we encountered, the above also appears to be true for failures caused by obfuscation. That is, most failures due to obfuscation appear to be caused by the same decompiler limitations that cause failures on unmodified code, rather than by a deliberate attempt to prevent decompilation. This is corroborated by the fact that, in several cases where a decompiler fails due to obfuscation, at least one other decompiler succeeds 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. The absence of this kind of "unbreakable" control-flow obfuscation suggests that, at least in theory, near-perfect decompilation is attainable for Android apps, with improvements to current decompiler implementations.

It should be noted, however, 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. [17], 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, we observed that there were negligible benefits from using other decompilers in addition to the best-

performing one (jadx), if one's intent was to decompile *every* method of an app. This is because the much higher failure rates of the non-native decompilers made it very likely that they would have at least a few failures on the apps for which jadx was unable to decompile all methods. However, if several independently developed decompilers with *similar* mean failure rates were used, the results might look different.

Despite the above findings, our results also showed that in 96% of cases where jadx failed to decompile a method, at least one of the other decompilers succeeded. This means that the overall success rate could be improved by employing several decompilers, despite one decompiler performing much better than the others.

B. Threats to Validity

The limitations of our methodology, which we have already discussed in Section III-B, 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 study is the representativeness of datasets, where our main concern is with the malware dataset. Firstly, the dataset is a few years old, meaning that it may no longer fully reflect, for example, obfuscation techniques used in present-day malware. Secondly, 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.

C. Future Work

The above threats to validity could both be partially addressed in future work. The matching accuracy of our approach could be improved 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. Another improvement to address in future work is to use a more recent malware dataset.

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 *quality* of recovered source code, as has already been done by others [18], [34], [35] for JVM bytecode decompilation.

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. Therefore, one direction for future work could be to integrate existing techniques [8], [25] for detecting third-party libraries into our analysis platform.

VII. Related Work

An early study of Java decompilation correctness was performed by Hamilton and Danicic [34] in 2009. Kostelanský and Dedera [35] performed a similar study in 2017, and concluded that the correctness of state-of-the-art decompilers had improved significantly since 2009. Harrand et al. [18] performed a large-scale study of 8 Java decompilers, in which they assessed both the syntactic and semantic correctness of recovered source code. Naeem et al. [36] proposed several metrics for measuring decompiler performance.

Jang et al. [19] 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 more advanced version of this concept was proposed by Harrand et al. [37], who use *meta-decompilation* to merge the results from several Java decompilers to improve overall decompiler effectiveness.

Dong et al. [38] 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.

Finally, a recent study by Hammad et al. [39] 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.

VIII. Conclusion

In this work we have presented the results of a large-scale study of the decompilation success rate of 4 different compilers on three large sets of Android apps. While the state-of-the-art Android decompiler jadx achieved a very low failure rate of only 0.02% 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. Finally, our empirical results and complementary manual investigation indicate that deliberate anti-decompilation obfuscation is not a major cause of decompilation failures in commercial or malicious apps. Instead, it appears that most failures 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.

References

[1] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed Android software," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1039–1055.

[2] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "StoryDroid: Automated generation of storyboard for Android apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 596–607.

[3] Z. Shan, I. Neamtiu, and R. Samuel, "Self-hiding behavior in Android apps: Detection and characterization," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 728–739.

[4] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, C. Raules, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, M. Grace *et al.*, "ATtention spanned: Comprehensive vulnerability analysis of AT commands within the Android ecosystem," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 273–290.

[5] F. Pauck, E. Bodden, and H. Wehrheim, "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*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 331–341.

[6] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of Android apps," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 358–369.

[7] L. Cen, C. S. Gates, L. Si, and N. Li, "A probabilistic discriminative model for Android malware detection with decompiled source code," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 4, pp. 400–412, 2015.

[8] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "LibD: Scalable and precise third-party library detection in Android markets," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 335–346.

[9] H. Wang, Y. Guo, Z. Ma, and X. Chen, "WuKong: A scalable and accurate two-phase approach to Android app clone detection." New York, NY, USA: Association for Computing Machinery, 2015.

[10] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *Trust and Trustworthy Computing*, S. Katzenbeisser, E. Weippl, L. J. Camp, M. Volkamer, M. Reiter, and X. Zhang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 291–307.

[11] A. Martín, H. D. Menéndez, and D. Camacho, "MOCDroid: multi-objective evolutionary classifier for Android malware detection," *Soft Computing*, vol. 21, no. 24, pp. 7405–7415, 2017.

[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *USENIX Security Symposium*, 2011.

[13] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, pp. 1–32, 2013.

[14] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM–software protection for the masses," in *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 2015, pp. 3–9.

[15] J.-T. Chan and W. Yang, "Advanced obfuscation techniques for Java bytecode," *Journal of systems and software*, vol. 71, no. 1-2, pp. 1–10, 2004.

[16] T.-W. Hou, H.-Y. Chen, and M.-H. Tsai, "Three control flow obfuscation methods for Java software," *IEE Proceedings-Software*, vol. 153, no. 2, pp. 80–86, 2006.

[17] V. Balachandran, D. J. Tan, V. L. Thing *et al.*, "Control flow obfuscation for Android applications," *Computers & Security*, vol. 61, pp. 72–93, 2016.

[18] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "The strengths and behavioral quirks of Java bytecode decompilers," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, pp. 92–102.

[19] H. Jang, B. Jin, S. Hyun, and H. Kim, "Kerberoid: A practical Android app decompilation system with multiple decompilers," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2557–2559.

[20] B. Pan. (2019) dex2jar. [Online]. Available: https://github.com/pxb1988/dex2jar/

[21] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003, pp. 290–299.

[22] J. Ming, D. Xu, L. Wang, and D. Wu, "Loop: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 757–768.

[23] (2020) F-Droid — Free and Open Source Android App Repository. [Online]. Available: https://www.f-droid.org/

[24] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current Android malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 2017, pp. 252–276.

[25] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in Android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 356–367.

[26] AppTornado GmbH. Free vs. paid Android apps. [Online]. Available: https://www.appbrain.com/stats/free-and-paid-android-applications

[27] (2020) jadx - Dex to Java decompiler. [Online]. Available: https://github.com/skylot/jadx

[28] L. Benfield. (2020) Cfr decompiler. [Online]. Available: https://www.benf.org/other/cfr/

[29] JetBrains s.r.o. (2020) Fernflower decompiler. [Online]. Available: https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine

[30] M. Strobel. (2020) Procyon decompiler. [Online]. Available: https://bitbucket.org/mstrobel/procyon/

[31] (2020) APKiD. [Online]. Available: https://github.com/rednaga/APKiD

[32] Google LLC. (2020) apkanalyzer. [Online]. Available: https://developer.android.com/studio/command-line/apkanalyzer

[33] B. Gruver. (2020) Smali/baksmali. [Online]. Available: https://github.com/JesusFreke/smali

[34] J. Hamilton and S. Danicic, "An evaluation of current Java bytecode decompilers," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2009, pp. 129–136.

[35] J. Kostelanský and Ĺ. Dedera, "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)*, 2017, pp. 1–6.

[36] N. A. Naeem, M. Batchelder, and L. Hendren, "Metrics for measuring the effectiveness of decompilers and obfuscators," in *15th IEEE International Conference on Program Comprehension (ICPC '07)*, 2007, pp. 253–258.

[37] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "Java decompiler diversity and its application to meta-decompilation," *Journal of Systems and Software*, vol. 168, p. 110645, 2020.

[38] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild," in *Security and Privacy in Communication Networks*. Springer International Publishing, 2018, pp. 172–192.

[39] M. Hammad, J. Garcia, and S. Malek, "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*, 2018, pp. 421–431.