DEMO: Reverse Engineering Android Apps with Code Coverage

Aleksandr Pilgun
SnT, University of Luxembourg
Luxembourg
aleksandr.pilgun@uni.lu

Abstract

Reverse engineering Android apps remains a critical and labor-intensive task, particularly for analyzing novel malware. Analysts typically begin with decompiled Java code using tools like JaDX and often must correlate it with runtime information gathered from dynamic analysis. In this work, we present JaDX-ACVTool, a plugin that bridges this gap by integrating code coverage information from ACVTool directly into JaDX-GUI. Our approach highlights Java methods executed during analysis, enabling security analysts to quickly identify and navigate runtime-relevant code paths.
Plugin repository: https://github.com/pilgun/jadx-acvtool

CCS Concepts

• **Security and privacy** → **Software reverse engineering**; *Mobile platform security*; • **Software and its engineering** → *Dynamic analysis.*

Keywords

Android, reverse engineering, code coverage, malware analysis

ACM Reference Format:
Aleksandr Pilgun. 2025. DEMO: Reverse Engineering Android Apps with Code Coverage. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3719027.3762169

1 Introduction

The Android ecosystem is friendly to app developers: anyone can create and publish an app on the Google Play Store or elsewhere on the Internet, which gives developers an opportunity to reach a wider audience. However, this openness also provides opportunities for malicious authors to create and distribute fraudulent applications.

In 2024, Google prevented 2.36 million policy-violating apps from being published on the Google Play Store. Additionally, 158 thousand developer accounts were banned [4]. Despite this serious effort, security companies keep reporting their findings on new threats, demonstrating that malicious actors are still ahead of defense measures [5]. According to recent reports, a single ad fraud campaign involving 331 applications reached more than 60 million users from 2024 through early 2025 [2].

CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3762169

Each successful fraud campaign essentially means bypassing well-tuned protections. When this happens, security analysts must analyze malicious samples, review protection mechanisms, and implement new detection rules. This process requires qualified professionals and involves significant manual effort. Unsurprisingly, manual analysis relies heavily on tools to decompile, disassemble, and analyze Android applications from a static analysis perspective. However, effective analysis often requires launching the app to observe specific behaviors and record additional runtime information [14].

Related work. Creators of Android apps can enjoy standard Android Studio tools when debugging and testing their applications, including the JaCoCo code coverage plugin, which can highlight Java code under test. However, the tooling for analysis of third-party applications without source code is **limited and less maintained**. Among the most commonly used and freely available decompilers are JD-GUI, JaDX-GUI, and Apktool.

JD-GUI is a popular Java decompiler that allows users to decompile Android apps when used with the dex2jar tool [3, 9]. JaDX-GUI is the most popular community-maintained open-source decompiler for Android apps that provides a user-friendly interface for navigating through decompiled Java code [12]. Apktool decompiles Android applications into `smali` representation, which is a more human-friendly assembly-like representation of Android bytecode.

Tools for third-party Android app tracing of executed code include BBoxTester, Cosmo [10], AndroLog [11], ACVTool [7], and WallMauer [1]. BBoxTester and Cosmo instrument decompiled Java code to measure code coverage. AndroLog instruments apps in Jimple representation and logs execution of code units into logcat. ACVTool is an instrumentation tool that measures code coverage and highlights executed code in `smali` representation. ACVTool is actively maintained and has demonstrated a high success rate [8]. WallMauer is an instrumentation tool to measure code coverage at the bytecode level with an approach very similar to ACVTool's. However, ACVTool is **the only tool** reliably producing user-friendly HTML reports to highlight executed code. ACVTool highlights app execution in `smali` representation, which is not as convenient for visual inspection compared to decompiled Java code in JaDX-GUI. In contrast, JaDX-GUI does not allow highlighting executed code in any way similar to ACVTool. Thus, we seek to enhance manual analysis in JaDX-GUI with code coverage information produced by ACVTool.

Contribution. In this work, we present the plugin for JaDX-GUI that integrates ACVTool reports. The plugin enables highlighting of executed methods in Java representation directly within JaDX-GUI, and provides direct navigation for analysts from Java classes to the corresponding `smali` code highlighted by ACVTool. As a running example, we analyze one sample of RuMMS malware to demonstrate the benefits of using the JaDX-ACVTool plugin.

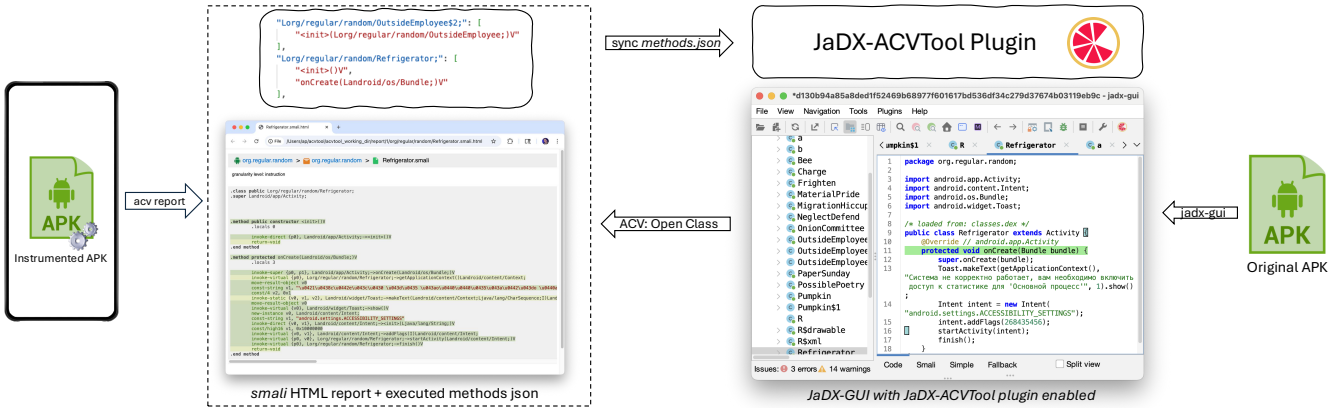

Figure 1: The JaDX-ACVTool workflow

2 JaDX-ACVTool Plugin

Our plugin bridges ACVTool reports with JaDX-GUI interface to enhance manual analysis of Android apps. It enables highlighting of executed methods in Java directly within JaDX-GUI. This integration also allows analysts to easily navigate from Java code to the corresponding `smali` class report highlighted by ACVTool.

2.1 Plugin Design

JaDX allows extending its functionality with plugins. This requires creating a Java project with a JaDX dependency and deriving the main class from the JadxPlugin class. Each plugin can be installed directly from JaDX-GUI interface by providing the JAR file or plugin ID when published in the official JaDX plugin list [13]. Plugins can also implement additional options that become available in JaDX-GUI settings.

Workflow. Figure 1 demonstrates a typical workflow of our plugin. To enable Java code highlighting, we must first perform the standard ACVTool workflow: instrument the APK, install it on a device, test the app, and generate the code coverage report [8]. ACVTool generates code coverage reports with `smali` representation highlighted in HTML format together with a list of executed methods stored in a JSON file (`methods.json`).

To start app analysis reinforced with method highlighting, an analyst opens the APK file using JaDX-GUI, then navigates to the Java class of interest. The plugin automatically synchronizes JaDX Java classes with `smali` classes in the ACVTool working directory (configurable in the plugin settings). As the analyst navigates through the application classes in JaDX-GUI, they receive immediate visual feedback on the executed methods.

2.2 Implementation

We implement our approach as a plugin for JaDX-GUI to easily extend its functionality without disrupting the existing JaDX codebase, since JaDX is diligently maintained by the community. However, we are constrained by the JaDX core APIs for traversing Java code, which are not optimal for matching code structures between Java and `smali` representations.

JaDX-GUI relies on the RSyntaxTextArea library for displaying decompiled Java code. RSyntaxTextArea is an open-source syntax highlighting text component for Java Swing applications.

In our implementation, we parse Java code of the currently opened class line by line to identify method declarations. We then match these declarations with the method signatures provided by the ACVTool `methods.json` file. Corresponding methods are highlighted in the Java representation using capabilities provided by RSyntaxTextArea. Since RSyntaxTextArea does not allow us to directly reference individual Java statements, we highlight only method declarations in Java representation, which is sufficient for our manual analysis.

Our plugin also provides a toolbar button and a context menu action to navigate from Java code to the corresponding ACVTool `smali` class report. This allows analysts to examine the executed instructions in `smali` representation highlighted by ACVTool.

3 Running Example

For our running example, we selected a sample from the RuMMS malware family, which is a well-known SMS fraud malware that abuses accessibility services and includes remote control capabilities [15]. We perform **manual analysis** of the app using JaDX-GUI, ACVTool, and our plugin to demonstrate the benefits of the JaDX-ACVTool plugin.

Static analysis in JaDX-GUI. The sample app has the package name `org.regular.random` and contains only 21 classes and 65 methods. The app is lightly obfuscated by replacing class, method, and variable names with random words and letters. Despite its small size, it is not immediately obvious how the malware operates when opened in JaDX-GUI.

The `AndroidManifest.xml` file lists *seven* entry points, including *four* activities, *four* services, and *three* receivers. Upon opening the main activity class, we find that the `onCreate` method only calls another empty method before the activity terminates itself. Another activity launches a misleading toast notification and opens Android accessibility settings to trick the user into enabling accessibility services. The third activity requests additional permissions. The fourth activity terminates itself immediately after being launched.

```java
    public static void a(Context context, int i, String str) {
        Intent intent = new Intent(context, (Class<?>) Pumpkin.class);
        intent.putExtra("body", str);
        intent.putExtra("app", "http://quostpeopls.com/uw0itx1tzypjsbwv/
        intent.putExtra("type", i);
        context.startService(intent);
    }
```

Figure 2: A method highlighted in Java

```
.method public static a(Landroid/content/Context;ILjava/lang/String;)V
        .locals 3

        const-string v0, "http://quostpeopls.com/uw0itx1tzypjsbwv/index.php"
        new-instance v1, Landroid/content/Intent;
        const-class v2, Lorg/regular/random/Pumpkin;
        invoke-direct {v1, p0, v2}, Landroid/content/Intent;-><init>(Landroic
        const-string v2, "body"
        invoke-virtual {v1, v2, p2}, Landroid/content/Intent;->putExtra(Ljava
        const-string v2, "app"
        invoke-virtual {v1, v2, v0}, Landroid/content/Intent;->putExtra(Ljava
        const-string v0, "type"
        invoke-virtual {v1, v0, p1}, Landroid/content/Intent;->putExtra(Ljava
        invoke-virtual {p0, v1}, Landroid/content/Context;->startService(Land
        return-void
.end method
```

Figure 3: A method highlighted in `smali`

The `Application` class contains more complex logic, including declaring an additional thread that runs in the background to collect phone characteristics. It also checks if the app is the default SMS app and verifies other alarm and power capabilities to keep itself active when the app is hidden in the background.

Dynamic analysis. We instrument the app using ACVTool and launch it on our Android device. Upon execution, we observe that the app does not display any visible activity, though it fires a misleading toast message and launches the Accessibility settings. The app keeps running in the background without any user interaction.

Hybrid analysis. Upon enabling our plugin in JaDX-GUI, we observe executed methods highlighted in the Java representation. We can utilize full JaDX-GUI functionality to navigate between app classes, yet immediately recognize which methods were executed since they are highlighted.

ACVTool calculated 30% instruction coverage for our simple app launch, with 27 out of 65 methods actually executed during app launch. In real-world applications, we expect a dramatic decrease in the surface area for manual analysis, as only a small fraction of methods are executed during app launch, as we pointed out in our previous work [6].

By examining highlighted Java methods in this experiment, we confirm that the app indeed collected phone characteristics, registered to listen for SMS messages, created and acquired a wake lock to keep the device awake, set up an alarm triggering an intent every 60 minutes, and attempted to connect to a C2 server. Notably, all of this occurred in the background without any user interaction. We could also observe that HTTP requests did not succeed, which is expected since the C2 server is no longer available.

Thus, we enhanced our manual analysis in two ways. First, the JaDX-ACVTool plugin helped us focus on a reduced code surface for analysis while keeping the code readable and organized for navigation in Java. Second, we could confidently confirm malicious behavior since the highlighted code proves it was executed.

4 Conclusion

We reported on a novel plugin for JaDX-GUI that bridges the gap between static and dynamic analysis of Android applications by integrating ACVTool runtime code execution information directly into the decompiled Java code view. This integration addresses a significant limitation in current Android malware analysis workflows, where analysts must manually correlate execution traces with Java code navigation.

JaDX-ACVTool plugin highlights executed methods in Java representation, which dramatically reduces the analysis surface from potentially thousands of methods to only those actually executed during runtime. In our RuMMS malware case study, this reduced the analysis scope from 65 methods to just 27 executed methods, while maintaining the readability and navigation advantages of Java code over `smali` representation.

Acknowledgments

This work was supported in part by the Luxembourg National Research Fund (FNR), grant reference 18154263 (UNLOCK).

References

[1] Michael Auer, Iván Arcuschin Moreno, and Gordon Fraser. 2024. Wallmauer: Robust code coverage instrumentation for android apps. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*. ACM/IEEE, Lisbon, Portugal, 34–44.

[2] BitDefender. 2025. *BitDefender Report on Ad Fraud Campaign.* Retrieved 2025-01-15 from https://www.bitdefender.com/en-us/blog/labs/malicious-google-play-apps-bypassed-android-security

[3] Java Decompiler. 2015. JD-GUI. Retrieved 2025-07-11 from https://github.com/java-decompiler/jd-gui

[4] Google. 2025. *How we kept the Google Play & Android app ecosystems safe in 2024.* Retrieved 2025-01-29 from https://security.googleblog.com/2025/01/how-we-kept-google-play-android-app-ecosystem-safe-2024.html

[5] AO Kaspersky Lab. 2025. *Kaspersky has discovered SparkKitty: a new Trojan spy on App Store and Google Play.* Retrieved 2025-06-23 from https://www.kaspersky.com/about/press-releases/kaspersky-has-discovered-sparkkitty-a-new-trojan-spy-on-app-store-and-google-play

[6] Aleksandr Pilgun. 2020. Don't Trust Me, Test Me: 100% Code Coverage for a 3rd-party Android App. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Singapore, 375–384.

[7] Aleksandr Pilgun, Olga Gadyatskaya, Stanislav Dashevskyi, Yury Zhauniarovich, and Artsiom Kushniarou. 2018. An effective android code coverage tool. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Toronto, ON, Canada, 2189–2191.

[8] Aleksandr Pilgun, Olga Gadyatskaya, Yury Zhauniarovich, Stanislav Dashevskyi, Artsiom Kushniarou, and Sjouke Mauw. 2020. Fine-grained code coverage measurement in automated black-box Android testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–35.

[9] pxb1988. 2015. Tools to work with android .dex and java .class files. Retrieved 2025-07-11 from https://github.com/pxb1988/dex2jar

[10] Andrea Romdhana, Mariano Ceccato, Gabriel Claudiu Georgiu, Alessio Merlo, and Paolo Tonella. 2021. Cosmo: Code coverage made easier for android. In *2021 14th IEEE conference on software testing, verification and validation (ICST)*. IEEE, Porto de Galinhas, Brazil, 417–423.

[11] Jordan Samhi and Andreas Zeller. 2024. AndroLog: Android Instrumentation and Code Coverage Analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. ACM, Porto de Galinhas, Brazil, 597–601.

[12] Skylot. 2015. JaDX Dex to Java decompiler. Retrieved 2025-07-11 from https://github.com/skylot/jadx

[13] Skylot. 2024. *JaDX Plugin Guide.* Retrieved 2025-07-11 from https://github.com/skylot/jadx/wiki/Jadx-plugins-guide

[14] Thomas Sutter, Timo Kehrer, Marc Rennhard, Bernhard Tellenbach, and Jacques Klein. 2024. Dynamic security analysis on android: A systematic literature review. *IEEE Access* 12 (2024), 57261–57287.

[15] Liu Wang, Haoyu Wang, Ren He, Ran Tao, Guozhu Meng, Xiapu Luo, and Xuanzhe Liu. 2022. MalRadar: Demystifying android malware in the new era. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–27.