OPEN ACCESS

KING'S College LONDON

King's Research Portal

DOI:
10.1007/978-3-032-30699-9_13

Document Version
Peer reviewed version

Link to publication record in King's Research Portal

Citation for published version (APA):
Even-Mendoza, K., Dakhama, A., & Berger, H. (2026). ApkFuzz: Search-Based Fuzzing for Android APK Vulnerability. *Lecture Notes in Computer Science*, 6. Advance online publication. https://doi.org/10.1007/978-3-032-30699-9_13

Citing this paper
Please note that where the full-text provided on King's Research Portal is the Author Accepted Manuscript or Post-Print version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version for pagination, volume/issue, and date of publication details. And where the final published version is provided on the Research Portal, if citing you are again advised to check the publisher's website for any subsequent corrections.

General rights
Copyright and moral rights for the publications made accessible in the Research Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognize and abide by the legal requirements associated with these rights.

•Users may download and print one copy of any publication from the Research Portal for the purpose of private study or research.
•You may not further distribute the material or use it for any profit-making activity or commercial gain
•You may freely distribute the URL identifying the publication in the Research Portal

Take down policy
If you believe that this document breaches copyright please contact librarypure@kcl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 17. Nov. 2026

ApkFuzz: Search-Based Fuzzing for Android APK Vulnerability Discovery

Karine Even-Mendoza[1][0000-0002-3099-1189], Aidan Dakhama[2][0009-0002-7318-7964], and Harel Berger[3][0000-0002-6388-4160]

[1] King's College London, London, UK. karine.even_mendoza@kcl.ac.uk
[2] University of Edinburgh, Edinburgh, UK. adakhama@ed.ac.uk
[3] Ariel University, Israel. harelb@ariel.ac.il

Abstract. While most work focuses on executable code, `AndroidManifest.xml`, serves as the primary entry point for application metadata and permission logic. Yet, it is often overlooked, making it a high-trust, low-scrutiny attack surface. We present ApkFuzz, an AFL-based fuzzer with bit-level mutations targeting the encoded region of `AndroidManifest.xml` in the APK file. We have identified 9 issues that cause crashes and enable *Denial-of-Service* (DoS) attacks in core tools, including Google's `apksigner` and the widely used research tool `DroidBot`.

Keywords: SBSE · Android · Fuzzing · AFL · APK · DoS

1 Introduction

The Android operating system, as the most widely deployed mobile platform, remains a frequent target of attacks on both its malware detection systems [2,3,4,13] and application infrastructure [7,15,19]. Despite recent advances using GenAI for improved contextual analysis [12], these approaches still struggle to capture the full structural complexity of real-world applications. As a result, the robustness of security analysis tools and frameworks, like the dynamic analyzer DroidBot [14] and *apksigner* utility, remains critical.

An *Android application package* (APK) contains both executable code and structured metadata defining its behavior and execution context. APKs are protected by digital signatures and integrity checks, so any modification of their contents breaks the original signing state. Among these components, the manifest file, `AndroidManifest.xml` acts as the blueprint of the application, containing essential metadata such as versioning, permissions, and intent filters. Despite its importance, the manifest file is often treated as a static configuration and receives significantly less attention in automated analysis than executable code. Even small manipulations can alter how an application is interpreted and executed, affecting dependencies, operational behaviour, and security properties. Blackbox fuzzing and manual testing are not well-suited to exploring the fault-injection search space, as they are not directed toward the manifest-specific region within APKs. While guided fuzzing is a natural candidate approach, byte-level modifications in the manifest still induce a large, sparse search space.

Contributions. We design ApkFuzz, an AFL-based, coverage-guided fuzzer with custom bit-level mutations on APK files, targeting the `AndroidManifest.xml` region while largely preserving package structure and signature consistency. We evaluate ApkFuzz on *apksigner*, *DroidBot*, and *ADB*, and identify 9 issues in *DroidBot* and *apksigner*. Three issues have been responsibly disclosed, while the rest are withheld pending developer response.

We make the following contributions:

- ApkFuzz, a novel search-based vulnerability discovery system designed to identify flaws in Android security and analysis tools.
- The discovery of a vulnerability in *apksigner* that results in crashes, and
- Two vulnerabilities in *DroidBot* leading to DoS via infinite loops.

Availability. Our code, bug reports, and fuzzing data and scripts are in [8].

2 Methodology

We model our approach fuzzing as a coverage-guided search over APK binaries, restricted to the manifest region. However, a direct application of AFL with APKs as seeds is not feasible in practice due to seed limitations: Most fuzzers are designed to handle diverse input types, but typically perform best on inputs in the KB range and degrade as inputs grow toward larger file sizes. APKs are significantly larger and more complex (often hundreds of KB to several MB), making them incompatible with many off-the-shelf fuzzing pipelines. Hence, we retain AFL's evolutionary loop (coverage feedback, queue, energy scheduling) and modify **i)** the input representation (APK byte archives) and **ii)** the mutation operator (region-constrained bit-flips over manifest offsets). We evaluate using crash/hang discovery and branch coverage using a short fuzzing campaign (§3).

Corpus of Binaries & System Under Test. We downloaded small- to medium-sized APK binaries from the internet, listed on Table 1, including their manifest start and end offsets, and the effective bytes available for fuzzing, to ensure size diversity of seeds in the input corpus. We did not use the manifest as a seed because the manifest is embedded in a hash-signed APK archive, extracting it invalidates the signature and breaks the APK's execution context. The *system under test* (SUT) is software taking an APK binary as input (not just a manifest), e.g., an APK validator or simulator.

Fuzzing. Our fuzzing approach for APKs is based on a region-aware binary mutator. The current design targets APKs, using the region information in Table 1. The *input corpus* is all seeds in Table 1. The *output corpus* contains the fuzzed APK binaries. Unlike standard fuzzing campaigns that target a single application, ApkFuzz mutates multiple APK binaries within a single campaign, enabling shared feedback to improve scheduling, coverage, and mutation selection. We mutate APK archives directly, without unpacking them into their uncompressed form. Our *custom mutator* targets a specific region of interest by operating on byte intervals defined by APK archive offsets, instead of uniformly mutating the entire APK. At each fuzzing iteration, ApkFuzz selects a seed from

Table 1. Test Inputs in the Initial Corpus of ApkFuzz (March 2026)

APK	Start Offset	Size (bytes)	End Offset
ademar-textlauncher-10	4602	1037	5639
com-gcstar-scanner-1	2463	617	3080
com-gcstar-viewer-12	3998	984	4982
com-java-SmokeReducer-1	51861	727	52588
com-martinmimigames-simplefileexplorer-20	17415	1230	18645
com-omgodse-notally-56	1079935	3513	1083448
com-page-bizzle-2	6190	913	7103
com-trianguloy-clipboardeditor-9	8866	1256	10122
dev-pranav-applock-221	1621968	3270	1625238
dubrowgn-wattz-20	453684	1377	455061
eu-hxreborn-remembermysort-300	39565	665	40230
F-Droid	7401341	8921	7410262
me-velc-devqs-3	6606	1256	7862
org-bc-bd-mrwhite-6	3601	630	4231
org-billthefarmer-editor-194	112532	1520	114052
weather-shalltry-group	9226856	7345	9234201

the queue (i.e., a valid APK archive), then uniformly samples a byte within the target region and applies a bit-flip mutation. ApkFuzz stores seeds triggering failures or timeouts in a dedicated crash/hang directory for further analysis.

Why Fuzzing the APK Manifest? We focus on the `AndroidManifest.xml` region in this paper. This region governs core aspects of Android application behavior, including permissions, components, and configuration metadata, essential for application installation and execution. The exact byte range corresponding to this region is identified separately for each APK in the corpus. Automatic identification and exploration of other APK regions are left for future work.

Implementation Details. We use AFL++, a fork [10] of the American Fuzzy Lop (`AFL`) fuzzer [20]. We implemented ApkFuzz in C and Python [8]. The custom mutator is implemented in C and includes validation of table lookups and APK file access for robust evaluation. To address AFL++'s large input files limitation, we adopt a SearchSYS-like strategy [5,6], loading APK binaries only in the custom mutator to keep the AFL-visible seeds small.

3 Evaluation

Research Questions. We ask:

- ➤ **RQ1.** How does region-constrained, bit-level mutation impact the fault-revealing capabilities of fuzzing on Android analysis pipelines?
- ➤ **RQ2.** How does ApkFuzz impact coverage on *DroidBot* within an 8-hour fuzzing budget using valid APKs as seed inputs?

Experimental Setup. We used ApkFuzz to fuzz the `DroidBot` codebase using APK binaries as input seeds. We selected `DroidBot` as the main fuzzing target because it is pure Python, so Py-AFL instrumentation is simple. Post-fuzzing, following a methodology similar to GrayC [9], we tested: **1)** *DroidBot* : an automated Android app testing tool that interacts with APKs on a simulator or physical device; **2)** *ADB* (Android Debug Bridge): a command-line tool used to communicate with an Android device or emulator. It supports installing APKs,

running shell commands, collecting logs, and controlling the device during testing; and **3)** *apksigner* : a tool for signing Android APKs and verifying their signatures, ensuring applications are properly authenticated before installation or testing. We fuzzed with no fixed schedule on bare metals: a MacBook Pro 2024 with Apple M2, 16 GB RAM, running macOS 26.4 (25E246); and CloudLab nodes c6420 and c6320. Instrumentation used python-afl for DroidBot.

RQ1 Results. During our evaluation, we discovered 9 issues in `apksigner` and `DroidBot`. We reported a representative issue to each bug reporting system with full details on the ApkFuzz's GitHub. We discuss them below.

Bug #1: *Parser Robustness Vulnerabilities in apksigner Triggered by Inconsistent APK Metadata in Manifest:* During evaluation, we identified a family of parser robustness issues in Google's official `apksigner` utility, which does not gracefully reject APKs whose `AndroidManifest.xml` entry has a size mismatch (compressed or uncompressed) between its Local File Header and Central Directory. Instead, it throws exceptions, e.g.,

```
Exception in thread "main" com.android.apksig.apk.ApkFormatException:
    Failed to read AndroidManifest.xml
  at ...
    com.android.apksigner.ApkSignerTool.main(ApkSignerTool.java:88)
Caused by: com.android.apksig.zip.ZipFormatException: Data Descriptor
    presence mismatch between Local File Header and Central Directory for
     entry AndroidManifest.xml. LFH: false, CD: true
...
```

and the entire process crashed. Although signature verification can fail with an error message for corrupted APKs, these inputs exposed inconsistent and non-robust error handling in the manifest parsing stage, potentially leading to a DoS condition. In contrast, standard archiving tools such as `unzip` extract the same files without issue. We reported the "Compressed size mismatch between Local File" variant to the developers.

Bug #2: *Infinite Installation Retry Loop in DroidBot:* During APK deployment in DroidBot, installation failures triggered by terminal ADB errors (e.g., `CHUNKED_SHA256` digest mismatches indicating integrity failures or unhandled `AndroidManifest.xml` parser exceptions) caused the framework to hang.

```
[CONNECTION] DroidBotIme is enabled and connected.
Please wait while installing the app...
adb: failed to install /users/kevenmen/fuzzing_run_1/corpus//
    flashlight_1774742581_869270785.apk: Failure [
    INSTALL_PARSE_FAILED_UNEXPECTED_EXCEPTION: Failed to parse /data/app/
    vmdl1499327455.tmp/base.apk: AndroidManifest.xml]
Please wait while installing the app...
....
Please wait while installing the app...
```

`DroidBot` entered an unbounded loop of installation attempts, incorrectly assuming the application was not yet installed, rather than handling the `INSTALL_PARSE_FAILED` status returned by ADB. This caused cascading failures in the analysis environment, including disconnection of components such as Logcat and the UserInputMonitor, and ultimately a system-wide hang. This represents a DoS vulnerability in automated analysis pipelines, where a malformed APK can neutralize the testing infrastructure. We reported `ALL_PARSE_FAILED_NO_CERTIFICATES` and `ALL_PARSE_FAILED_UNEXPECTED_EXCEPTION` variants to the developers.

Bug #3: *DroidBot Crashed during Cleanup when no Emulator was Up* as it failed to handle gracefully missing or unavailable emulator devices.

```
INFO:Device:disable minicap on emulator
adb: device 'emulator-5554' not found
Traceback (most recent call last):
 ...
subprocess.CalledProcessError: Command '['adb', '-s', 'emulator-5554',
    ...
Traceback (most recent call last):
  ...
During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  ...
  File "/users/kevenmen/droidbot/droidbot/droidbot.py", line 188, in stop
    self.device.tear_down()
AttributeError: 'NoneType' object has no attribute 'tear_down'
```

Instead of exiting cleanly with an informative error, the framework continued execution and triggered a secondary exception as uninitialized (None) `device` object led to an unhandled `AttributeError`, which crashed *DroidBot*. This reflects inconsistent exception handling during initialization and shutdown phases.

RQ2 Results. We report the final DroidBot fuzzing campaign (Python mode), which ran for 30 hours. The resulting corpus contained 29 seeds, achieving 30.68% coverage (41.38% new edges). Six seeds were marked as favored. The primary seed, `weather-shalltry-group`, generated 2,267 executions and resulted in 3 crashes, 3 hangs, and 10 queue entries; `F-Droid` was the second most favored seed with 263 executions, 1 crash and 1 queue entry. Since prior fuzzing runs also uncovered bugs in other packages, like `com-gcstar-viewer-12` and `ademar-textlauncher-10`, these results suggest that, despite non-trivial coverage, further tuning of mutation strategies and seed selection is needed.

4 Conclusion

Android testing focuses on runtime code execution and communication like [11,18], which focus on graybox and cross-language fuzzing to test closed-source native libraries and vendor services, and like [1,16], which steer application execution paths to uncover inter-process communication flaws and environment-triggered malicious behaviours. Hence, parsing and metadata layers such as `AndroidManifest.xml` remain a largely unexplored attack surface. To address this, we introduce ApkFuzz, a search-based region-aware approach to test the robustness of Android tooling.

We utilize *Fuzzing*, which has been successfully used to find bugs in systems software [5,17,20]. We build on prior search-based fuzzing work on binary executables [5,6] that mutates C-generated executables to fuzz system simulators. We use the same coverage-based fitness settings [20,6] and a similar seed management mechanism [6], but redefine the search space to APK binaries with region-constrained mutation and APK-aware seed handling, forming our primary design contributions beyond standard AFL.

By applying structure-preserving, bit-level mutations to the manifest region, ApkFuzz bypasses signature validation and exposes deep parser-level logic flaws. Our evaluation revealed 9 vulnerabilities, including critical DoS conditions in Google's *apksigner* and *DroidBot*. These results expose fragility in how dynamic analysis pipelines handle core configuration metadata. Future work will automate vulnerable APK byte regions discovery, extend to other archive formats such as iOS IPAs, and incorporate more rigorous fuzzing setups with repeated runs.

References

1. Askar, A., Fleischer, F., Kruegel, C., Vigna, G., Kim, T.: Malintent: Coverage guided intent fuzzing framework for android (2025)
2. Berger, H., Dvir, A., Mariconti, E., Hajaj, C.: Breaking the structure of mamadroid. Expert Systems with Applications **228**, 120429 (2023)
3. Berger, H., Hajaj, C., Mariconti, E., Dvir, A.: Crystal ball: From innovative attacks to attack effectiveness classifier. IEEE Access **10**, 1317–1333 (2021)
4. Bostani, H., Moonsamy, V.: Evadedroid: A practical evasion attack on machine learning for black-box android malware detection. Computers & Security (2024)
5. Dakhama, A., Even-Mendoza, K., Langdon, W.B., Menéndez, H.D., Petke, J.: Enhancing search-based testing with llms for finding bugs in system simulators. Autom. Softw. Eng. **32**(2) (2025). https://doi.org/10.1007/s10515-025-00531-7
6. Dakhama, A., Even-Mendoza, K., Langdon, W., Menendez, H., Petke, J.: Searchgem5: Towards reliable gem5 with search based software testing and large language models. In: Search-Based Software Engineering. pp. 160–166 (2024)
7. Datta, S., et al.: Cross-boundary mobile tracking: Exploring java-to-javascript information diffusion in webviews. In: NDSS 2026
8. Even-Mendoza, K., Dakhama, A., Berger, H.: Artifact of apkfuzz: Search-based fuzzing for android apk vulnerability discovery (2026). https://doi.org/10.5281/zenodo.19476523
9. Even-Mendoza, K., Sharma, A., Donaldson, A.F., Cadar, C.: GrayC: Greybox fuzzing of compilers and analysers for C. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 1219–1231. ISSTA 2023. https://doi.org/10.1145/3597926.3598130
10. Fioraldi, A., et al.: AFL++ : Combining incremental steps of fuzzing research. In: WOOT USENIX Workshop 2020
11. Iannillo, A.K., et al.: Chizpurfle: A gray-box android fuzzer for vendor service customizations. In: ISSRE 2017. https://doi.org/10.1109/ISSRE.2017.16
12. Kurlandski, L., Berger, H., Pan, Y., Wright, M.: Beyond raw bytes: Towards large malware language models. In: NDSS 2026
13. Li, D., Li, Q.: Adversarial deep ensemble: Evasion attacks and defenses for malware detection. IEEE TIFS (2020). https://doi.org/10.1109/TIFS.2020.3003571
14. Li, Y., Yang, Z., Guo, Y., Chen, X.: Droidbot: a lightweight ui-guided test input generator for android. In: ICSE-C '17. https://doi.org/10.1109/ICSE-C.2017.8
15. Luo, T., et al.: Attacks on webview in the android system. In: ACSAC 2011. pp. 343–352. https://doi.org/10.1145/2076732.2076781
16. Rasthofer, S., Arzt, S., Triller, S., Pradel, M.: Making malory behave maliciously: Targeted fuzzing of android execution environments. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). pp. 300–311. IEEE (2017)
17. V. T. Pham, M.B., Roychoudhury, A.: AFLNET: A greybox fuzzer for network protocols. In: ICST 2020. https://doi.org/10.1109/ICST46399.2020.00062
18. Xiong, H., Dai, Q., Chang, R., Qiu, M., Wang, R., Shen, W., Zhou, Y.: Atlas: Automating cross-language fuzzing on android closed-source libraries. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 350–362 (2024). https://doi.org/10.1145/3650212.3652133
19. Xu, H., Yao, M., Zhang, R., Dawoud, M.M., Park, J., Saltaformaggio, B.: DVa: Extracting victims and abuse vectors from android accessibility malware. In: 33rd USENIX Security Symposium (USENIX Security 24)
20. Zalewski Michal: Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt (Retrieved April 21, 2023)