APK/AAB Anti-Decompilation Research Report

Authors: Sarvesh Aadhitya, Shyamalavannan, Karthik

Date: July 30, 2025

Classification: Technical Research Report

Index

Executive Summary

This report documents comprehensive research into anti-decompilation techniques for Android applications, focusing on methods to prevent reverse engineering tools like JADX and BundleTool from successfully parsing APK and AAB files. The research explored multiple attack vectors targeting ZIP structure manipulation and DEX file corruption while maintaining application installability and runtime functionality.

Key Findings:

- APK locking techniques proved successful against JADX decompiler
- AAB locking techniques failed against BundleTool's robust validation
- The Android Runtime's strict verification mechanisms limit effective obfuscation approaches
- Simple malformation attempts are insufficient against modern decompilation tools

1. Introduction

Android applications distributed as APK (Android Package Kit) or AAB (Android App Bundle) files are fundamentally ZIP archives containing compiled bytecode, resources, and metadata. Reverse engineering these applications poses significant security risks including intellectual property theft, credential extraction, and application tampering.

This research investigated various anti-decompilation techniques designed to break common reverse engineering tools while preserving application functionality. The primary challenge lies in creating malformations sophisticated enough to confuse parsers while remaining valid according to Android Runtime specifications.

2. Research Methodology and Attempted Techniques

2.1 Comprehensive Technique Evaluation

The following table summarizes all researched anti-decompilation techniques and their effectiveness:

Technique	Breaks JADX	Breaks BundleTool	App Installs	Status
EOCD Corruption	❌	❌	✅	Failed
Dual Central Directory	✅	❌	❌	Failed
Fake Compression Flag	✅	❌	✅	**Partial Success**
Unaligned APK	✅	❌	❌	Failed
CN-Protect Shell Injection	✅/❌	❌	❌	Failed
Extra Data After EOCD	✅	❌	❌	Failed
Header Shuffling	✅	✅	❌	Failed
Decoy Manifest/DEX Swap	✅	❌	❌	Failed
Unicode Method Mangling	✅	❌	✅	**Partial Success**
DEX Bytecode Injection	❌	❌	❌	Failed

2.2 Failed Approaches Analysis

Most attempted techniques failed due to one of three primary reasons:

1. **Runtime Verification Failures**: The Android Runtime's Dalvik/ART verifier rejected malformed bytecode, causing VerifyError exceptions
2. **Installation Failures**: Package Manager validation rejected corrupted ZIP structures
3. **Tool Resilience**: Modern decompilation tools proved robust against simple tampering attempts

3. Successful Implementations

3.1 APK Locking Success: Local File Header Corruption

The APK locking approach successfully broke JADX parsing through Local File Header (LFH) corruption while maintaining application installability.

Technique Details:

- Target: Compression method field in LFH of classes*.dex files
- Method: Overwrite 2-byte compression field with invalid value (0xFFFF)
- Result: JADX fails to extract DEX files due to unknown compression method
- Limitation: Android Package Manager ignores compression flag for uncompressed entries

3.2 AAB Locking Failure: BundleTool Resilience

The AAB locking approach using Smali-level obfuscation failed against BundleTool's validation mechanisms.

Attempted Techniques:

- Unicode method name mangling using full-width characters
- Junk code injection (NOP and GOTO instructions)
- Decoy class insertion
- Result: BundleTool's robust validation detected and rejected modifications

4. Technical Implementation Details

4.1 APK Locking Implementation (lfh_corrupt.py)

```
#!/usr/bin/env python3
import os
import sys
import subprocess

# Constants
KEYSTORE_FILE = "debug.keystore"
KEY_ALIAS = "androiddebugkey"
STORE_PASS = "android"

def corrupt_local_header_compression(apk_path, output_path):
    """
    Corrupts the Local File Header compression method for classes*.dex files
    """
    with open(apk_path, 'rb') as f:
        data = bytearray(f.read())

    # Local File Header signature
    lfh_signature = b'\\x50\\x4B\\x03\\x04'

    i = 0
    while i < len(data) - 30:  # Minimum LFH size
        if data[i:i+4] == lfh_signature:
            # Extract filename length and extra field length
            filename_len = int.from_bytes(data[i+26:i+28], 'little')
            extra_len = int.from_bytes(data[i+28:i+30], 'little')

            # Extract filename
            filename_start = i + 30
            filename_end = filename_start + filename_len
            filename = data[filename_start:filename_end].decode('utf-8', errors='ignore')

            # Check if this is a classes*.dex file
            if filename.startswith('classes') and filename.endswith('.dex'):
                print(f"Corrupting LFH for: {filename}")
                # Corrupt compression method (offset +8, 2 bytes)
                data[i+8:i+10] = b'\\xFF\\xFF'  # Invalid compression method

                # Move to next potential LFH
                i += 30 + filename_len + extra_len
            else:
                i += 1

    # Write corrupted APK
    with open(output_path, 'wb') as f:
        f.write(data)

def sign_apk(apk_path, output_path):
    """
    Signs the APK using apksigner
    """
    cmd = [
        'apksigner', 'sign',
        '--ks', KEYSTORE_FILE,
        '--ks-key-alias', KEY_ALIAS,
```

```
        '--ks-pass', f'pass:{STORE_PASS}',
        '--out', output_path,
        apk_path
    ]

    result = subprocess.run(cmd, capture_output=True, text=True)
    if result.returncode != 0:
        print(f"Signing failed: {result.stderr}")
        return False
    return True

def main():
    if len(sys.argv) != 2:
        print("Usage: python3 lfh_corrupt.py <input.apk>")
        sys.exit(1)

    input_apk = sys.argv[1]
    base_name = os.path.splitext(os.path.basename(input_apk))[0]

    # Step 1: Corrupt LFH
    corrupted_apk = f"{base_name}_corrupted.apk"
    corrupt_local_header_compression(input_apk, corrupted_apk)

    # Step 2: Sign corrupted APK
    final_apk = f"{base_name}_signed.apk"
    if sign_apk(corrupted_apk, final_apk):
        print(f"Successfully created: {final_apk}")
        os.remove(corrupted_apk)  # Clean up intermediate file
    else:
        print("Failed to sign APK")

if __name__ == "__main__":
    main()
```

This script implements an APK tamper-protection method by intentionally corrupting the Local File Header compression method for all `classes*.dex` files inside the APK. It achieves this by scanning the APK's binary structure for Local File Headers, identifying the target DEX files, and overwriting the compression method field with an invalid value (`0xFFFF`). This corruption disrupts standard ZIP parsers, causing many decompilation tools to fail. The APK is then re-signed using the Android debug keystore to ensure it remains installable. While effective against basic static analysis, the method can be bypassed by directly parsing the APK's Central Directory.

Technical Background

An APK file is essentially a **ZIP archive** with a specific structure.

Each file in the ZIP has:

- A **Local File Header (LFH)** – contains metadata such as compression method, file name length, and extra fields.
- A **Central Directory (CD)** – contains an index of files.
- The **file data** itself.

Compression method field in LFH is a 2-byte value at offset `+8` from the start of the LFH.

- `0x00` → Stored (no compression)
- `0x08` → Deflate compression
- Changing this to an invalid value (like `0xFFFF`) **confuses ZIP parsers** that rely on LFH.

4.2 AAB Locking Implementation (hybrid.py)

```
#!/usr/bin/env python3
import os
import sys
import subprocess
import zipfile
import tempfile
import shutil
import re

# Constants
BUNDLETOOL_JAR = "bundletool-all-1.18.1.jar"
BAKSMALI_JAR = "baksmali.jar"
SMALI_JAR = "smali.jar"
KEYSTORE_FILE = "debug.keystore"
KEY_ALIAS = "androiddebugkey"
STORE_PASS = "android"

def inject_junk_and_confuse(smali_dir):
    """
    Injects junk code and applies Unicode method name mangling
    """
    # Unicode character mapping for method name obfuscation
    unicode_map = {
        'a': 'ａ', 'b': 'ｂ', 'c': 'ｃ', 'd': 'ｄ', 'e': 'ｅ',
        'f': 'ｆ', 'g': 'ｇ', 'h': 'ｈ', 'i': 'ｉ', 'j': 'ｊ',
        'k': 'ｋ', 'l': 'ｌ', 'm': 'm', 'n': 'ｎ', 'o': 'ｏ',
        'p': 'ｐ', 'q': 'ｑ', 'r': 'ｒ', 's': 'ｓ', 't': 'ｔ',
        'u': 'ｕ', 'v': 'ｖ', 'w': 'w', 'x': 'ｘ', 'y': 'ｙ',
        'z': 'ｚ'
    }

    def mangle_method_name(name):
        """Convert ASCII method names to full-width Unicode"""
        return ''.join(unicode_map.get(c.lower(), c) for c in name)

    # Process all .smali files
    for root, dirs, files in os.walk(smali_dir):
        for file in files:
            if file.endswith('.smali'):
                file_path = os.path.join(root, file)

                with open(file_path, 'r', encoding='utf-8') as f:
                    content = f.read()

                # Apply method name mangling
                content = re.sub(
                    r'\\.method\\s+([^;]+\\s+)(\\w+)(\\([^)]*\\)[^;]*;)',
                    lambda m: f'.method {m.group(1)}{mangle_method_name(m.group(2))}{m.group(3)}',
                    content
                )

                # Inject junk code before method endings
                junk_code = """
nop
```

```
    nop
    goto :junk_label

    :junk_label
    nop
"""
            content = re.sub(
                r'(\\s+)\\.end method',
                lambda m: f'{junk_code}{m.group(1)}.end method',
                content
            )

            with open(file_path, 'w', encoding='utf-8') as f:
                f.write(content)

    # Add decoy class
    decoy_class_content = """
.class public LCancerClass;
.super Ljava/lang/Object;

.method public constructor <init>()V
    .registers 1
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
    return-void
.end method

.method public ｊｕｎｋＭｅｔｈｏｄ()V
    .registers 2
    const-string v1, "This is junk"
    nop
    nop
    goto :junk_loop

    :junk_loop
    nop
    return-void
.end method
"""

    decoy_path = os.path.join(smali_dir, 'CancerClass.smali')
    with open(decoy_path, 'w', encoding='utf-8') as f:
        f.write(decoy_class_content)

def process_aab(aab_path):
    """
    Main processing function for AAB files
    """
    base_name = os.path.splitext(os.path.basename(aab_path))[0]

    with tempfile.TemporaryDirectory() as temp_dir:
        # Extract AAB
        aab_extract_dir = os.path.join(temp_dir, 'aab_extracted')
        with zipfile.ZipFile(aab_path, 'r') as zip_ref:
            zip_ref.extractall(aab_extract_dir)

        # Find and process DEX files
        for root, dirs, files in os.walk(aab_extract_dir):
            for file in files:
                if file.endswith('.dex'):
                    dex_path = os.path.join(root, file)
```

```
            print(f"Processing DEX: {dex_path}")

            # Disassemble DEX to Smali
            smali_dir = os.path.join(temp_dir, f'smali_{file}')
            os.makedirs(smali_dir, exist_ok=True)

            baksmali_cmd = [
                'java', '-jar', BAKSMALI_JAR, 'd',
                dex_path, '-o', smali_dir
            ]

            result = subprocess.run(baksmali_cmd, capture_output=True, text=True)
            if result.returncode != 0:
                print(f"Baksmali failed: {result.stderr}")
                continue

            # Inject obfuscation
            inject_junk_and_confuse(smali_dir)

            # Reassemble Smali to DEX
            smali_cmd = [
                'java', '-jar', SMALI_JAR, 'a',
                smali_dir, '-o', dex_path
            ]

            result = subprocess.run(smali_cmd, capture_output=True, text=True)
            if result.returncode != 0:
                print(f"Smali assembly failed: {result.stderr}")
                continue

    # Rebuild AAB
    modified_aab = os.path.join(temp_dir, f'{base_name}_modified.aab')
    with zipfile.ZipFile(modified_aab, 'w', zipfile.ZIP_DEFLATED) as zip_ref:
        for root, dirs, files in os.walk(aab_extract_dir):
            for file in files:
                file_path = os.path.join(root, file)
                arcname = os.path.relpath(file_path, aab_extract_dir)
                zip_ref.write(file_path, arcname)

    # Generate APKS using BundleTool
    apks_file = os.path.join(temp_dir, 'universal.apks')
    bundletool_cmd = [
        'java', '-jar', BUNDLETOOL_JAR,
        'build-apks',
        '--bundle', modified_aab,
        '--output', apks_file,
        '--mode', 'universal',
        '--ks', KEYSTORE_FILE,
        '--ks-key-alias', KEY_ALIAS,
        '--ks-pass', f'pass:{STORE_PASS}'
    ]

    result = subprocess.run(bundletool_cmd, capture_output=True, text=True)
    if result.returncode != 0:
        print(f"BundleTool failed: {result.stderr}")
        return False

    # Extract universal APK
    with zipfile.ZipFile(apks_file, 'r') as zip_ref:
        zip_ref.extract('universal.apk', temp_dir)
```

```
# Align and sign final APK
aligned_apk = os.path.join(temp_dir, 'aligned.apk')
zipalign_cmd = ['zipalign', '-v', '4',
                os.path.join(temp_dir, 'universal.apk'), aligned_apk]
subprocess.run(zipalign_cmd, capture_output=True)

# Final signing
final_apk = f"{base_name}_final_signed.apk"
sign_cmd = [
    'apksigner', 'sign',
    '--ks', KEYSTORE_FILE,
    '--ks-key-alias', KEY_ALIAS,
    '--ks-pass', f'pass:{STORE_PASS}',
    '--out', final_apk,
    aligned_apk
]

result = subprocess.run(sign_cmd, capture_output=True, text=True)
        if result.returncode == 0:
            print(f"Successfully created: {final_apk}")
            return True
        else:
            print(f"Final signing failed: {result.stderr}")
            return False

def main():
    if len(sys.argv) != 2:
        print("Usage: python3 hybrid.py <input.aab>")
        sys.exit(1)

    aab_path = sys.argv[1]
    if not os.path.exists(aab_path):
        print(f"File not found: {aab_path}")
        sys.exit(1)

    success = process_aab(aab_path)
    if not success:
        print("AAB processing failed")
        sys.exit(1)

if __name__ == "__main__":
    main()
```

5. Decompiler Analysis

5.1 JADX Parsing Behavior

JADX (Java Decompiler for Android) operates as a comprehensive DEX-to-Java decompiler with the following characteristics:

Parsing Process:

1. ZIP structure validation and extraction
2. DEX file format verification
3. Dalvik bytecode analysis and type inference
4. Java code reconstruction with syntax highlighting

Vulnerabilities Identified:

- Sensitive to Local File Header corruption in ZIP structures
- Custom ZIP reader implementation provides some tampering resistance
- Fails when encountering invalid compression methods
- Generally robust against simple obfuscation techniques

Limitations:

- Cannot handle severely malformed DEX files that would also crash Android Runtime
- Built-in deobfuscator can reverse basic identifier renaming
- Resilient against most structural ZIP manipulations

5.2 BundleTool Parsing Behavior

BundleTool demonstrates significantly more robust validation mechanisms:

Parsing Process:

1. Comprehensive AAB structure validation
2. Strict adherence to Android App Bundle specifications
3. Digital signature verification
4. Bytecode integrity checks

Strengths:

- Resistant to Smali-level obfuscation attempts
- Validates Unicode method names and rejects suspicious patterns
- Detects junk code injection through static analysis
- Employs comprehensive error checking throughout processing pipeline

Why AAB Locking Failed:

- BundleTool's validation process detected Unicode method name mangling as suspicious
- Junk code injection patterns were identified and rejected
- The tool's robust error handling prevented successful manipulation

6. Technical Challenges and Limitations

6.1 The Android Runtime Verification Paradox

The fundamental challenge in Android anti-decompilation lies in the Android Runtime's strict verification requirements:

Dalvik/ART Verifier Requirements:

- Type consistency across all execution paths
- Proper register initialization and usage
- Correct instruction sequencing (invoke-* followed by move-result-*)
- 4-byte alignment for pseudo-instructions
- Exception handlers must begin with move-exception

The Core Paradox:
Any bytecode modification sophisticated enough to break decompilers risks triggering VerifyError exceptions, rendering the application non-functional. This creates an inherent tension between obfuscation effectiveness and application stability.

6.2 DEX File Structure Complexity

DEX files exhibit complex interdependencies that make casual modification extremely difficult:

Critical Dependencies:

- Header contains checksums and offsets to all other sections
- String tables require precise indexing and MUTF-8 encoding

- Method definitions contain pointers to bytecode locations
- Class definitions must maintain inheritance ordering

Modification Challenges:

- Single byte changes can cascade through multiple sections
- Checksum and signature validation must be maintained
- Specialized tools like dexmod are required for safe modifications

7. Conclusions and Future Recommendations

7.1 Research Outcomes

This comprehensive research into APK/AAB anti-decompilation techniques yields several critical insights:

Successful Approaches:

- APK locking through Local File Header corruption proves effective against JADX
- Technique maintains application installability while breaking decompilation
- Simple, targeted approach more effective than complex obfuscation

Failed Approaches:

- AAB locking failed due to BundleTool's robust validation
- Complex Smali-level obfuscation detected and rejected
- Most ZIP structure manipulations either break installation or prove ineffective

7.2 Why Comprehensive APK/AAB Locking Is Not Feasible

Based on extensive research and experimentation, achieving comprehensive anti-decompilation protection for Android applications faces insurmountable technical barriers:

Technical Impossibility Factors:

1. **Runtime Verification Constraints:** The Android Runtime's strict verification mechanisms prevent most effective obfuscation techniques from functioning without causing application crashes.
2. **Tool Evolution:** Modern decompilation tools like JADX and BundleTool continuously evolve to counter anti-analysis techniques, creating an unsustainable arms race.
3. **Platform Requirements:** Android's open ecosystem demands adherence to strict packaging and validation standards that limit effective obfuscation opportunities.
4. **Fundamental Contradiction:** Effective anti-decompilation requires code that is simultaneously interpretable by the Android Runtime but incomprehensible to static analysis tools—a technically contradictory requirement.

7.3 Final Assessment

While partial success was achieved in breaking JADX parsing for APK files, the failure to effectively protect AAB files and the fundamental limitations imposed by Android's architecture demonstrate that comprehensive application protection through static obfuscation alone is not technically feasible. Security strategies must evolve beyond simple anti-decompilation techniques toward more sophisticated, multi-layered approaches that acknowledge the inherent limitations of static protection mechanisms.

The research confirms that the Android ecosystem's design principles, while beneficial for application compatibility and security, inherently limit the effectiveness of anti-reverse engineering techniques. Future security research should focus on runtime protection mechanisms and server-side architectures rather than pursuing increasingly complex static obfuscation approaches that offer diminishing returns against modern analysis tools.