APKARMOR: Low-Cost Lightweight Anti-Decompilation Techniques for Android Apps

Jiayang Liu*†, Yanjie Zhao*†, Pengcheng Xia†, and Haoyu Wang‡†
†Huazhong University of Science and Technology, China
{jiayangliu, yanjie_zhao, xpc357, haoyuwang}@hust.edu.cn

***Abstract*—Android app security is a critical concern for the software industry, with companies investing significantly in protecting their intellectual property from reverse engineering attacks. While commercial protection tools exist to prevent decompilation and unauthorized code access, they pose substantial challenges for businesses: high licensing costs ranging from thousands to tens of thousands of dollars annually, significant performance overhead that impacts user experience and app ratings, and increased app size that affects download rates. These limitations particularly burden small to medium-sized enterprises and independent developers, creating an urgent industry need for cost-effective protection solutions.**

To address these challenges, we propose a novel file format-based anti-decompilation strategy that systematically exploits structural vulnerabilities in APK files. Building upon this strategy, we have developed APKARMOR, a lightweight and cost-effective anti-decompilation framework that exploits inherent vulnerabilities in popular reverse engineering tools. Through systematic analysis, we first identified critical weaknesses in common decompilation tools' parsing mechanisms and structural assumptions. Based on these findings, we developed seven mutation-based protection strategies that deliberately trigger these vulnerabilities by introducing specific structural anomalies into APK files and the AndroidManifest.xml. These methods include Countermeasures against Dirty Code and Corrupted Payloads (CACoP), Pseudo-Encryption (PE), Using Unknown Compression Method (UUCM), Unavailable Magic Value (UMA), Modify the Offset Field in stringChunk (MOFS), and Dirty Bytecode Replacement of "Android" (DRA). We evaluated our exploitation strategies through extensive experiments on 100 randomly selected Android apps, testing against the latest versions of three widely used decompilation tools: JADX (v1.5.1), APKTool (v2.11.0), and Androguard (v4.1.2). Our results demonstrate that PE and DRA achieved complete protection by successfully exploiting vulnerabilities present in all tested tools. MOFS, UUCM, and UNV effectively exploited weaknesses in APKTool and Androguard's parsing mechanisms.

I. INTRODUCTION

Mobile apps have become integral to modern business operations, with Android dominating over 70% of the global mobile OS market as of 2024 [1], representing a vast ecosystem of apps that generate billions in revenue. However, this widespread adoption has attracted malicious actors, leading to a significant rise in reverse engineering attempts targeting apps with valuable intellectual property or payment systems [2], [3]. These threats pose severe risks to businesses, especially small to medium-sized enterprise development teams(SMEs) lacking dedicated security resources.

The global mobile application security market was valued at USD 5.2 billion in 2023 and is projected to grow at a remarkable compound annual growth rate (CAGR) of over 22% from 2024 to 2032, reflecting the escalating demand for robust security solutions in an increasingly digital-first economy [4]. Despite the growing investment in mobile application security, reverse engineering attacks targeting Android applications continue to escalate at an alarming rate. Notably, SMEs remain disproportionately vulnerable, representing the majority of successful compromise incidents. This heightened susceptibility is primarily attributable to their constrained access to sophisticated protection mechanisms compared to larger organizations with dedicated security resources [5].

Although commercial protection technologies including code obfuscation, runtime encryption, and VM-based protection represent the industry's standard defense mechanisms, their significant performance overhead and implementation costs create substantial barriers for SMEs. Empirical evidence indicates these security measures introduce considerable resource demands, with obfuscation techniques alone typically expanding APK sizes by 20-40% while simultaneously increasing CPU execution overhead by 8-12% [6]. The premium features' recurring subscription fees further exacerbate financial burdens, particularly for small-to-medium enterprises Table I. These technical and financial constraints render current approaches neither accessible nor scalable for SMEs operating under strict performance requirements and tight budget constraints.

TABLE I: Pricing for Android app protection services.

Protection Service Type	Service Provider	Price Range
ProGuard/Obfuscation Service	Various Vendors	$500 - $5000/year
Commercial Protection Suite	DexGuard, GuardSquare	$2000 - $10,000+/year
Anti-Tampering Protection	Security Vendors	From $3000/year

To address these limitations, we present a novel anti-reverse engineering strategy that fundamentally differs from conventional approaches. Our innovation lies in Android APK file format mutation, which strategically targets inherent vulnerabilities in reverse engineering toolchains rather than modifying

* Jiayang Liu and Yanjie Zhao contributed equally to this work.
† The full name of the author's affiliation is Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.
‡ Haoyu Wang (haoyuwang@hust.edu.cn) is the corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00262

application logic or introducing resource-intensive protection mechanisms. Through meticulous investigation, we have identified a critical yet overlooked attack surface: the parser-level error tolerance exhibited by mainstream analysis tools such as Apktool and JADX when processing non-standard or deliberately malformed file structure elements. By exploiting these parsing vulnerabilities, our approach delivers robust protection while maintaining minimal performance overhead and requiring no licensing fees.

Building upon these findings, we developed APKARMOR, an automated protection system that seamlessly integrates file format mutation strategies into standard development workflows. APKARMOR implements seven effective modification techniques through a configurable build plugin, requiring minimal developer intervention while ensuring consistent application of the most effective mutations for each target scenario. Our extensive evaluation on 100 commercial applications demonstrates that specific modification strategies achieve complete success in preventing decompilation across popular reverse engineering tools, while maintaining negligible impact on application performance and user experience.

The main contributions of this work are as follows:

- We provide a comprehensive analysis of parsing vulnerabilities in widely-used decompilation tools, documenting how specific structural modifications in APK files can effectively disrupt their operation across multiple aspects including DEX file parsing, resource handling, and manifest processing.
- We present an innovative file format mutation-based protection strategy that systematically identifies seven effective modification techniques capable of disrupting reverse engineering tools while maintaining minimal performance overhead (less than 5% increase in app size with negligible runtime impact).
- Through extensive experiments on 100 commercial apps across various categories, we demonstrate APKARMOR's effectiveness against popular reverse engineering tools, showing that our approach achieves 100% success in preventing decompilation for specific modification strategies while exhibiting selective effectiveness depending on the particular target tool.

II. BACKGROUND

A. Widely-Used Android Decompilers

Decompilation tools are important in reverse engineering, used to convert compiled binary code back into source code. For Android apps, commonly used decompilation tools include JADX, APKTool, and Androguard. These tools each have their own characteristics and scope of the app, helping security researchers, developers, and reverse engineering enthusiasts gain a deeper understanding of the internal structure and working principles of apps.

JADX [7] is a graphical Java decompilation tool used to view the source code of JAR files. It works in conjunction with Dex2Jar to decompile JAR files converted by Dex2Jar and display the original Java source code. JD-GUI parses Java bytecode, converts it into equivalent Java source code, and displays it in a tree structure, making it convenient for users to browse and analyze. APKTool [8] is a powerful Android app decompilation tool that can decompile APK files into Smali code and resource files, supporting modification and repackaging of APK files. APKTool parses the resources and bytecode of APK files, converting them into readable Smali code and resource files. Users can modify the Smali code and resource files and use APKTool to repackage and generate new APK files. Androguard [9] is an open-source Android reverse engineering tool that integrates multiple functions, including decompilation, static analysis, and dynamic analysis. It supports multiple file formats and analysis methods, making it suitable for complex reverse engineering tasks. Androguard parses the bytecode structure and resource files of DEX files, providing various analysis tools and interfaces to support deep code analysis and vulnerability mining.

Modern decompilation tools typically follow a three-stage workflow. First, the target file format is parsed to extract bytecode and resources—for instance, APKTool deciphers APK structures while Dex2Jar processes DEX bytecode. The extracted bytecode then undergoes conversion to intermediate representations, such as Java bytecode (Dex2Jar) or Smali instructions (APKTool). Finally, these intermediates are transformed into analyzable outputs like Java source code (JADX) or Python objects (Androguard).

The decompilation ecosystem features specialized tools with distinct tradeoffs. Dex2Jar excels at rapid DEX-to-JAR conversion but struggles with heavily obfuscated bytecode. JD-GUI provides straightforward JAR decompilation with clean output, though its deobfuscation capabilities remain limited. APKTool supports full APK repackaging and resource editing, albeit with occasional compatibility issues when handling protected apps. For advanced reverse engineering, Androguard offers comprehensive analysis features at the cost of steeper learning curves and configuration complexity.

B. Anti-Decompilation Techniques

In Android app development, anti-decompilation techniques [10] are common protective measures aimed at preventing the app's code from being reverse-engineered and cracked. Traditional anti-decompilation techniques mainly include hardening, encryption, and obfuscation. App hardening protects the app's code by making it difficult to decompile and reverse engineer. Common hardening methods include code injection, instruction rearrangement, and virtualization. Code injection involves injecting malicious or useless code into the app [11], making the decompiled code difficult to read and understand. Instruction rearrangement changes the execution order of the code, preventing reverse engineering tools from generating the correct source code [12]. Virtualization converts part of the code into virtual machine instructions and embeds a virtual machine in the app to execute these instructions, thereby increasing the complexity and difficulty of the code [13].

Encryption technology encrypts the app's code, resources, strings [14], [15], [16], [17], and class definitions [18], making them impossible to decompile without proper decryption. Common encryption methods include encrypting DEX files, resource files, strings, and class structures. When the app starts, these encrypted components are decrypted and loaded into memory for execution. By securing these elements, encryption effectively protects the integrity of the code and resources, preventing them from being decompiled, analyzed, or stolen.

Obfuscation technology obfuscates the app's code, making the decompiled code difficult to understand and analyze [19], [20], [21]. Common obfuscation methods include variable [22] and method name obfuscation, control flow obfuscation [23], [24], and code folding. Variable and method name obfuscation replace the variable and method names in the code with meaningless names, making the decompiled code difficult to understand. Control flow obfuscation changes the control flow of the code, making the decompiled code difficult to analyze. Code folding folds some logical structures in the code, making the decompiled code more complex and difficult to understand.

III. Preliminary Study

In this section, we present a comprehensive preliminary analysis of anti-decompilation techniques through three phases (as shown in Figure 1): intelligence gathering from security forums and technical communities, empirical study on a large-scale Android app dataset, and in-depth case studies of selected samples exhibiting interesting decompilation resistance patterns. **This preliminary investigation, focusing on vulnerability discovery in decompilers, serves as a crucial foundation for developing effective strategies to bypass Android decompilation protections**.

A. Open-Source Intelligence Gathering

Compared to the traditional method of conducting exhaustive analyses on open-source or commercial software, we have decided to adopt a more efficient and targeted strategy to discover potential problem cases - by using crawler technology to collect intelligence from major security forums-StackExchange [25], Reddit_NetSec [26], XianZhi [27] and KanXue [28]. This method allows us to directly mine valuable information from the actual feedback of users and practitioners, enabling us to quickly locate potential issues that may arise in practical apps. Specifically, we targeted three widely-used open-source tools, JADX [7], APKTool [8] and Androguard [9]-on GitHub [29]. As the world's largest code hosting platform, GitHub brings together a large number of developers and users, making it a place to obtain first-hand user feedback and case studies. We carefully screened and collected problems from their release pages, including failed decompilation cases, software crashes, unexpected behaviors during analysis, etc.

This method was chosen based on the assumption that analysts primarily work with real, functional apps rather than corrupted samples or improper testing environments. This ensures that collected cases can be reliably reproduced on Android systems, enhancing the accuracy of our vulnerability analysis. We automated this process using a custom crawler program that regularly captures relevant issues and discussions from GitHub pages, storing them in a local database for subsequent categorization and analysis. The process can be described as follows:

TABLE II: Forum statistics including collection quantity and related content quantity.

Forum	Collection quantity	Quantity of relevant contents
Stack Exchange Security	1196	124
Reddit NetSec	1526	43
XianZhi	146	2
KanXue	795	85

Data collection for the security forums. We first extensively collect analysis cases of app parsing failures from major security forums. Although some of these cases may have lost their timeliness—meaning the issues they involve may have been fixed in the latest software versions, or the technical methods used are no longer mainstream—these cases still hold significant value. For our data collection, we specifically targeted a total of 3,363 entries from security forums and Android-related blogs, as illustrated in Table II, with a time span covering the past 2 to 3 years. This focused approach ensures that the collected cases are relevant.

Data processing and analysis. For the collected dataset, we first performed data preprocessing, which involved extracting relevant content from the blogs and handling missing or incomplete entries. Rather than extensive manual standardization, we employed a large language model (LLM), DeepSeek, to efficiently extract and summarize the key insights from the blog posts.

Manual analysis. Based on the LLM's classification results, we proceeded with in-depth manual analysis to validate and enrich our findings. In the manual review phase, we first filtered cases and discussions identified as potentially valuable by the LLM. Two authors then individually examined these selected cases, documenting and analyzing their specific characteristics and implications.

By analyzing the decompilation failure cases outlined in the "Reverse Engineering and Decompilation" category, we identified two anti-decompilation strategies that were effective at the time of the blog posts' publication. The first strategy takes advantage of the ZIP format by altering the APK's ZIP structure, which confuses decompilation tools without compromising the app's functionality. The second strategy targets vulnerabilities in decompilation tools' parsers by making specific adjustments to the AndroidManifest.xml format, exploiting weaknesses in how these tools handle the manifest file during decompilation.

B. Empirical Study-based Analysis

To complement our intelligence gathering efforts, we conducted a large-scale empirical study analyzing decompilation failures in Android apps. This empirical study aimed to

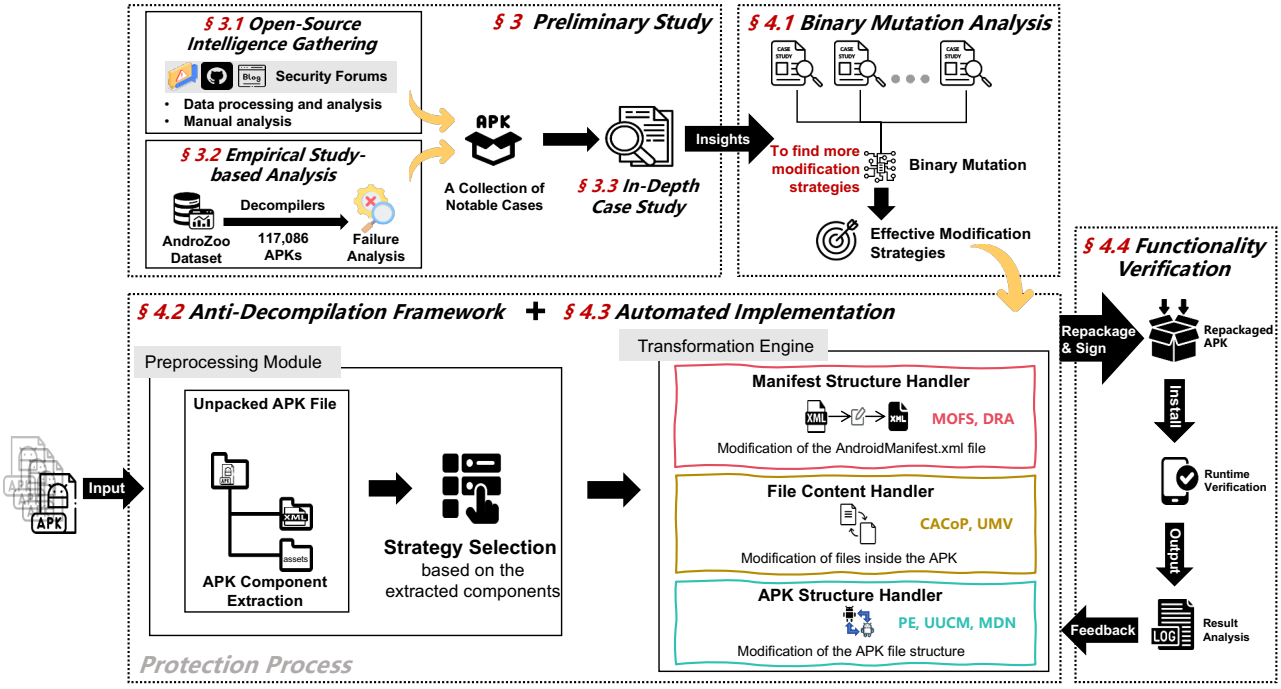

Fig. 1: Overview of our approach.

systematically identify and categorize various decompilation resistance patterns in real-world apps.

We leveraged the AndroZoo platform [30] to construct our dataset, collecting 117,086 Android APK samples. These samples represent diverse app categories and development practices, ensuring comprehensive coverage of potential decompilation scenarios. Using APKTool, a widely-adopted reverse engineering tool, we attempted to decompile all collected samples. The decompilation process revealed 1,580 failure cases, representing approximately 1.35% of the total samples.To distinguish the root causes of decompilation failures, we verified the integrity and functionality of the mutated APKs. The fact that all 1,580 corresponding APKs passed installation and normal runtime checks on Android devices provided definitive evidence that the APK files were not corrupted. It was therefore inferred that 1,575 of the observed failures were not due to file corruption or environmental issues, but could be attributed to the successful exploitation of specific vulnerabilities within the decompilers.

Through systematic examination of the failed cases, we identified three primary decompilation resistance patterns. The first pattern involves dirty code and corrupted payload techniques, where malformed code segments or payloads are deliberately inserted to trigger decompiler failures while maintaining app functionality. The second pattern exploits the ZIP format, manipulating the APK's ZIP structure to confuse decompilation tools while preserving the app's integrity. The third pattern focuses on AndroidManifest.xml format attacks, where strategic modifications to the manifest file format exploit parser vulnerabilities in decompilation tools.

This empirical study complements our intelligence gathering phase by providing quantitative evidence of decompilation resistance techniques in the wild. The findings establish a foundation for understanding current anti-decompilation practices and guide the development of more robust reverse engineering tools.

C. In-Depth Case Study

In this subsection, we conduct in-depth case studies based on the findings from § III-A and § III-B. Our analysis aims to identify specific causes of decompilation failures and understand how these findings can guide the development of anti-decompilation techniques. The distribution of failure causes is presented in Figure 2.

1) Exploiting Vulnerabilities based on the content of the APK file: **Case#1:** During the process of decompiling a large number of APK samples using APKTool (see § III-B), we encountered a specific error known as the "dex magic number mismatch", which resulted in the termination of the decompilation process of APKTool. Upon analyzing the samples, we identified the primary cause of this error to be APKTool's attempt to parse all files within the APK during the decompilation. Table III shows the parsing sequence of APKTool files. In one particular case, we found a sample in which the issue was exacerbated by the presence of a corrupt DEX file, which further complicates the parsing process. Malicious authors deliberately included a DEX file with an incorrect magic number in the resource files during the APK packaging process, causing APKTool to encounter an error and terminate

during the parsing stage. This discovery highlights the need for robust error handling and verification mechanisms in tools such as APKTool to gracefully manage such discrepancies.

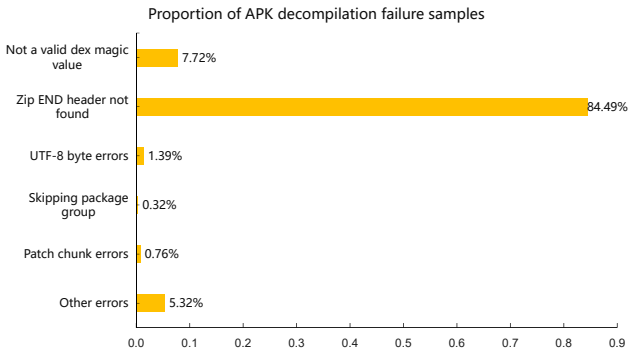

Fig. 2: Proportion of decompilation failures.

TABLE III: The order of parsing files of different types in APKs with APKTool.

Order	Resource Type	Description
1	AndroidManifest.xml	The configuration file of an app.
2	.dex	Bytecode produced by compiling Java or Kotlin code.
3	Signatures	File Signature in APK
4	Assets	Native resources required by the application.
5	Res	Android resource files.
6	Libs	Native libraries that the program depends on.

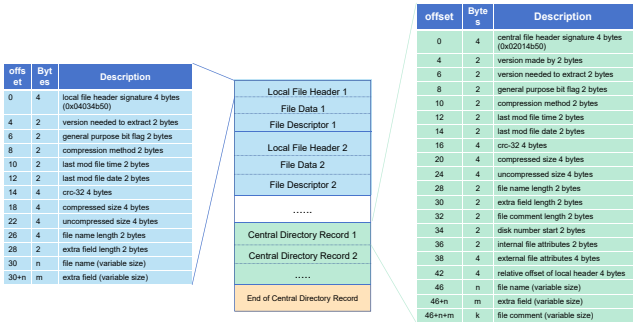

Fig. 3: `ZIP` data format.

2) Exploiting Vulnerabilities based on zip format: **Case#2:** When using tools such as JADX or APKTool to decompile apps, a specific error message is often encountered: "invalid CEN header (encrypted entry)". After thorough analysis, we have discovered that this is a method employed to combat decompilation by deliberately modifying the encryption flag in the ZIP format file (Figure 3). To further illustrate this point, we compared the data in the Central Directory of a normal app with that of an app employing this anti-decompilation technique. The results revealed that the only significant difference between the two lies in the value of the ninth byte. Specifically, the adversarial sample cleverly sets its FLAG value to indicate an encrypted format. This modification ultimately leads to the decompilation tool's inability to correctly identify and process the file, thereby achieving its purpose of resisting decompilation.

Case#3: During the process of decompiling datasets, we encountered instances where certain samples exhibited decompilation errors when using tools such as JADX or APKTool. The specific error message observed was "java.util.zip.ZipException: invalid CEN header". Upon comparative analysis of the adversarial samples and the original samples, we found no significant differences in their file structures. However, it was noted that the adversarial samples utilized a compression mode identified by number 7261. It is important to note that both JADX and APKTool rely on java.util.zip for decompressing APK packages, and unfortunately, this package does not support the decompression method corresponding to the number 7261. In contrast, the Android system employs the STORED compression method (designated with the number 0) as its default. If an unrecognized compression method is encountered, the Android system defaults to using the DEFLATE compression method (designated with the number 8). Consequently, these samples exploit an unknown compression method to induce a failure in decompression by the decompilation tools, while still allowing the Android system to parse them successfully.

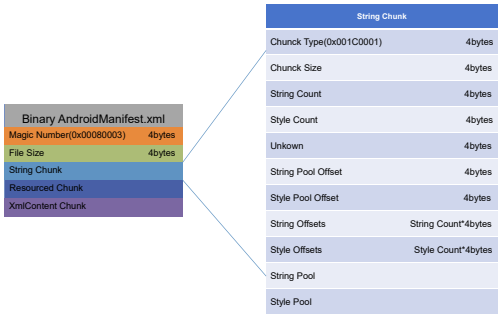

Fig. 4: `Androidmanifest.xml` data format.

3) Exploiting Vulnerabilities Based on `AndroidManifest.xml` File Format: **Case#4:** When using JADX or APKTool to decompile an app, an error message "Could not decode XML" may occur. Through analyzing the AndroidManifest.xml files of relevant APKs, we observed that, compared to normal AndroidManifest.xml files (as shown in Figure 4) that can be successfully decompiled, these special XML files modify the value of the String Offset field at the binary level. Specifically, they replace the normal offset values with abnormal ones, causing the decompiler to attempt to locate strings based on offsets that exceed the set data range. This leads to out-of-bounds access errors.

IV. Approach

Based on the vulnerabilities identified in the decompilation tools in § III, this section explores modification methods and uses them to attempt to counter existing decompilation tools. We begin by conducting a systematic binary mutation analysis to validate our previously identified modification strategies and explore additional structural variants through controlled mutations (§ IV-A). After determining the modification strategies (§ IV-B), we proceed with automated validation in § IV-C and § IV-D to assess the effectiveness of these strategies. Figure 1 illustrates the entire process of our approach.

A. Binary Mutation Analysis

Our research focuses on systematic binary mutation techniques to identify critical vulnerabilities in APK file structures that can disrupt decompilation processes. By carefully modifying specific binary-level features within both the APK container and its embedded AndroidManifest.xml file, we induce parsing failures in decompilation tools while ensuring the application remains fully functional on Android devices. The approach involves three primary mutation strategies: implementing pseudo-encryption through strategic modification of encryption flags, obfuscating the compression method by employing undefined format identifiers, and altering fundamental structural elements in the manifest file including string chunk offsets and magic number values. These precisely targeted modifications exploit the inherent differences between Android's forgiving runtime parsing behavior and the strict structural expectations of decompilation tools, creating reliable protection against reverse engineering attempts while maintaining complete compatibility with standard Android operation.

The observation metrics for our approach focus on three key aspects. The primary metric assesses whether the mutated APK maintains its functionality by successfully installing and executing on Android devices, which establishes the baseline validation for mutation effectiveness. The second metric evaluates whether common decompilers can successfully process the modified APK, where decompilation failures or errors indicate the effectiveness of our mutation strategy. The third metric involves a comparative analysis of the decompiled code before and after mutation, specifically examining how the mutations impact decompiler behavior and identifying instances of decompilation failures or errors. Through these comprehensive observation metrics, we can systematically evaluate the effectiveness of different mutation techniques in preventing successful decompilation while maintaining APK functionality.

Through the mutation results, we not only validated the four types of cases discussed in § III across a large dataset, but also discovered that, in addition to modifications targeting the ZIP or XML header fields, alterations can also be made to the contents of the AndroidManifest.xml file.

B. Anti-Decompilation Framework

The goal of the anti-decompilation methodology is to use a series of binary modification techniques to effectively prevent decompiler tools from analyzing and decompiling the modified APK file. The core of this methodology lies in systematically identifying and modifying the weaknesses of decompiler tools, ensuring that the protection strategy disrupts the decompilation process while allowing the modified APK to install and run normally on Android devices. Our approach combines two complementary analytical channels: empirical investigation (§ III) and mutation analysis (§ IV-A). The empirical investigation comprises two components: leveraging open-source intelligence from developer forums to identify decompiler vulnerabilities through reported issues (§ III-A), and analyzing decompilation failures across a large corpus of real-world applications (§ III-B). Mutation analysis focuses on thoroughly examining the binary structure of the APK file and the AndroidManifest.xml file to find key positions that can disrupt the behavior of decompiler tools. These mutations primarily target the structure of the APK file and the binary encoding of the AndroidManifest.xml file. By modifying key fields, we can effectively block decompilers from parsing and analyzing the APK.

In summary, our proposed anti-decompilation strategies include:

- **Countermeasures against Corrupted Payloads (CACoP)**: This strategy exploits parsing mechanism vulnerabilities in APK decompilation tools by strategically inserting damaged or malformed files at specific locations within the APK structure. These corrupted payloads effectively interfere with the decompilation process while maintaining the app's functionality, as the Android runtime can still correctly process these files.
- **Pseudo-encryption (PE)**: This technique involves modifying the encryption flag in the APK file structure to indicate that the content is encrypted, without actually performing any encryption. This manipulation exploits the assumption made by most decompiler tools that encrypted content requires decryption before processing, causing them to fail when attempting to parse the supposedly encrypted content. The Android system, however, processes the unencrypted content normally.
- **Using unknown Compression Method (UUCM)**: By altering the compression method identifier in the APK file to an unrecognized format, this strategy prevents decompiler tools from successfully decompressing the file contents. The modification targets specific fields in the ZIP file structure while ensuring the Android package installer can still correctly process the file using its more robust handling of compression methods.
- **Modify Disk Number (MDN)**: This approach involves manipulating the disk number fields within the APK's ZIP structure, which disrupts decompiler tools' ability to locate and extract resources. While decompilers rely on these values for file parsing, the Android runtime's more resilient im-

plementation allows it to handle such modifications without affecting app functionality.

- **Modify String Offset in StringChunk (MSOS)**: This strategy targets the string table structure within AndroidManifest.xml by modifying string offsets. These modifications prevent decompiler tools from correctly reading and interpreting critical manifest data, while the Android system's parsing mechanism remains unaffected due to its more robust implementation.
- **Unavailable Magic Value (UMV)**: This technique involves carefully altering magic values in the AndroidManifest.xml file structure. These values serve as format identifiers that decompiler tools use for validation. By modifying them to invalid but carefully chosen values, we prevent decompilation while maintaining compatibility with the Android runtime's more flexible parsing approach.
- **Dirty Bytecode Replacement of "Android" (DBR)**: This sophisticated approach replaces the standard "Android" string representation with complex, deliberately malformed bytecode sequences. The modified bytecode exploits differences between decompiler parsing logic and the Android runtime's processing, effectively disrupting decompilation while preserving normal execution.

The implementation of anti-decompilation strategies involves a systematic approach to modifying APK and AndroidManifest.xml files. Key fields within these files are identified as potential targets for modification, focusing on elements that play crucial roles in how decompiler tools parse the APK file. These strategies include modifying encryption flags, adjusting file compression methods, and altering the structure of key elements within the APK or manifest file, which can effectively interfere with the decompiler's ability to parse and analyze the file.

Once the modification targets are identified, binary modifications are carefully designed and applied to ensure the APK file maintains its functionality on Android devices. Each modified APK undergoes comprehensive automated testing, including installation and runtime validation on Android devices, to verify that core functionalities remain intact. The testing process ensures that the modifications do not introduce crashes, errors, or abnormal behaviors while effectively disrupting the decompilation process.

The effectiveness of our anti-decompilation strategies is evaluated through several critical metrics. First, the modified APK must successfully install and run on Android devices without compromising functionality. Second, the modifications should prevent decompilers from producing meaningful output, indicating successful protection against reverse engineering attempts. We systematically analyze automated test results and decompiler logs to track the performance of applied modifications, using this feedback to refine and optimize our protection techniques.

C. Automated Implementation

Based on the aforementioned methodology, we have developed a tool named APKARMOR. The core of the protection mechanism lies in the modular design of the tool, which provides high flexibility and automation capabilities. APKARMOR consists of multiple modules, including a modification strategy module, an automation processing module, a verification module, and a logging and feedback module. The modification strategy module stores and manages various anti-decompilation strategies; the automation processing module automates the entire process of analyzing and mutating APK files; the verification module ensures the functionality of mutated APKs; and the logging and feedback module records detailed information about the modification process to support optimization and troubleshooting.

Modification Strategies. Modification strategies are the core methods of the protection mechanism, aiming to interfere with decompilation tools by modifying APK file structures and key fields. These strategies include embedding deliberately crafted dex files into resource files to create parsing obstacles (CACoP), modifying encryption flags in the APK file structure to achieve pseudo-encryption (PE), using unknown compression methods, and adjusting key fields in the AndroidManifest.xml file (UUCM, MOFS). The goal of these modification strategies is to exploit weaknesses in the parsing mechanisms of decompilation tools, causing them to encounter unexpected issues during parsing and terminate the decompilation process.

Transformation Execution Workflow. The entire protection process is predominantly automated, from analyzing the target APK file to executing modification strategies and verifying the results. The initial APK analysis involves parsing the input APK file to extract its structure, identify suitable modification fields, and generate a detailed report highlighting potential mutation points. Following this, in the Strategy Selection phase, appropriate modification strategies are chosen from a predefined library based on APK characteristics and user configurations. During the modification execution phase, these strategies are applied using techniques such as CACoP, PE, UUCM, UNV, and DRA to enhance protection while preserving functionality. Finally, the Validation phase ensures the effectiveness of the modifications by testing the modified APK for successful installation and execution, and verifying its resistance to decompilation tools. Detailed logs of transformations and their impact are recorded, summarizing mutated fields, validation results, and observations on decompiler behavior.

Verification and Feedback. The final stage of the protection mechanism is verification and feedback, which ensures that mutated APK files function properly in real-world environments. The verification process includes automated installation and application launch, combined with runtime log analysis to confirm normal functionality and prevent user experience degradation caused by the protection mechanism. Detailed logs and reports generated by the tool not only document the entire modification process but also provide robust support for optimizing modification strategies and addressing new decompilation tools.

D. Functionality Verification

The functionality verification process is a critical step to ensure that mutated APK files can be properly installed and executed without issues. This process encompasses the design and execution of a robust verification mechanism, which includes verification standards, testing methods, and reporting and feedback procedures. The verification mechanism follows three key steps. First, the mutated APK is repackaged and signed as needed to meet Android's installation requirements. This step ensures that the structural integrity of the APK is preserved post-mutation. Next, the APK undergoes an installation test on an Android emulator or physical device, where the installation process is closely monitored for errors or warnings. Finally, the APK is executed to verify its runtime behavior, ensuring that it functions normally without crashes or abnormal behavior.

The verification standards focus on three primary aspects: installation success, functionality preservation, and the absence of crash information. Installation success requires that the APK installs smoothly on supported Android versions without any errors. Functionality preservation ensures that the core features of the APK remain operational after modification, although only successful installation and startup are mandatory for this verification process. Lastly, the runtime of the APK must be free from crashes or abnormal behaviors, demonstrating its stability after modification. To implement these standards, the testing methods rely heavily on automation. Automated scripts handle the APK installation across various Android versions and devices to verify compatibility. Additionally, automation tools are used to start the APK and monitor its runtime for crashes or irregularities. If the APK installs and runs without issues, it is deemed to have passed the verification process.

V. Evaluation

To validate the effectiveness of ApkArmor against various existing decompilation tools, we conducted comprehensive experiments on a set of Android apps. Our evaluation methodology is structured as follows: we first present our experimental setup in § V-A, including the selection criteria for test apps, detailed configuration of the testing environment, and our choice of representative decompilation tools commonly used in the industry. Subsequently, in § V-B, we provide a thorough analysis of the experimental results, demonstrating the effectiveness of each protection strategy against these tools through quantitative metrics and qualitative assessments. The evaluation focuses not only on the success rate of preventing decompilation but also on maintaining the functionality and performance of protected applications.

A. Evaluation Setup

Selected Decompilers. We selected three widely-used Android decompilation tools for our evaluation: JADX [7], APKTool [8], and Androguard [9]. These tools were chosen based on their popularity in the reverse engineering community and distinct technical characteristics.

App Dataset. We randomly sampled 100 Android apps from AndroZoo [30], which were originally published on the Google Play Store, to form our evaluation dataset. In order to ensure the accuracy of the experiment, we verify the original APK to ensure that it can be installed and run normally.

Running Environment. The experiments were conducted on a laptop running Windows 11, equipped with a 13th Gen Intel(R) Core(TM) i7-13700 featuring 16 cores and 32 threads operating at a base frequency of 2.10 GHz, 32 GB of RAM (2 modules of 16 GB each), and a 1GB NVMe SSD. Among the randomly selected 100 APKs, each mutated app was run in an Android 10 emulator environment.

B. Evaluation Results

In this section, we evaluate the effectiveness of the ApkArmor protection mechanisms by testing mutated APK files against multiple mainstream decompiler tools, including JADX (v1.5.1), APKTool (v2.11.0), and Androguard (v4.1.2). Using a range of modification methods, we assessed how well these modification strategies hinder the decompilation process. The results indicate that different modification strategies significantly impact the decompiler's ability to decompile, with varying success rates across tool versions. Below, we present the success rate of each protection method alongside the testing results and observed behavior for each decompiler version.

We first evaluated our modification strategies against JADX (v1.5.1, released on Nov 12, 2024). Table IV presents the results obtained after applying different modification strategies to 100 APKs. The modification strategies PE and DRA remained effective in the latest version, successfully preventing decompilation attempts. However, CACoP, UUCM, MDN, UMV, and MOFS became ineffective in this version. For APKTool (v2.11.0, released on January 15, 2024), the results shown in Table V indicate that modification strategies CACoP, PE, UUCM, UMA, MOFS, and DRA demonstrated strong effectiveness against the current version. However, MDN was no longer effective. Finally, results on Androguard (v4.1.2, released on May 31, 2024), as illustrated in Table VI, showed that modification strategies PE, UUCM, MOFS, and DRA were effective against the current version, while MDN and UMA were ineffective.

The experimental results indicate significant differences in the effectiveness of various modification methods across different decompiler tools. Specifically, PE and DRA methods performed well across all tested decompilers (JADX, APKTool, and Androguard), effectively preventing the decompilation of modified APK files. In contrast, the CACoP modification strategy is only effective for APKTool, while the MDN method failed to prevent decompilation across any of the tools, highlighting its limitations. Different versions of decompilers showed varying responses to the modification strategies. Strong modification stratigies like PE and DRA were better recognized and blocked by newer decompiler versions, while older versions were less effective, allowing successful decompilation. The effectiveness of CACoP, UUCM, and MOFS

largely depended on the decompiler version, suggesting that the success of these methods is closely tied to the tool's version. Thus, the update of decompiler tools and the adaptability of modification strategies are crucial, as newer versions may demonstrate greater resistance to these protection strategies.

TABLE IV: The anti-decompilation effectiveness of different strategies on JADX v1.5.1 (tested on 100 apps).

Modification Strategy	Anti-Decompilation Effectiveness	Last Effective Ver.
CACoP	No	1.4.0
PE	Yes	1.5.1
UUCM	No	1.5.0
MDN	No	1.4.7
UMV	No	1.4.7
MOFS	No	1.4.7
DRA	Yes	1.5.1

Note: "Yes" indicates that the protection method successfully prevented decompilation in all 100 tested apps, while "No" indicates the method failed to prevent decompilation in any test case. This notation applies to Table V and Table VI as well.

TABLE V: The anti-decompilation effectiveness of different strategies on APKTool v2.11.0 (tested on 100 apps).

Modification Strategy	Anti-Decompilation Effectiveness	Last Effective Ver.
CACoP	Yes	2.11.0
PE	Yes	2.11.0
UUCM	Yes	2.11.0
MDN	No	2.7.0
UMV	Yes	2.11.0
MOFS	Yes	2.11.0
DRA	Yes	2.11.0

TABLE VI: The anti-decompilation effectiveness of different strategies on Androgurad v4.1.2 (tested on 100 apps).

Modification Strategy	Anti-Decompilation Effectiveness	Last Effective Ver.
CACoP	No	3.4.0
PE	Yes	4.1.2
UUCM	Yes	4.1.2
MDN	No	3.4.0
UMV	No	3.4.0
MOFS	Yes	4.1.2
DRA	Yes	4.1.2

Technique	App Size Impact	Memory Impact
CACoP	controllable increase (10 100+ KB)	None
PE	Negligible (4 bytes)	None
UUCM	Negligible (2 bytes)	None
MDN	Negligible (4 bytes)	None
MSOS/UMV	Negligible (¡10 bytes)	None
DBR	Slight increase (1 10 KB)	Slight increase

TABLE VII: Impact of Different File Format Mutation Techniques on App Size and Memory Usage

Based on the experimental results, PE and DRA are the most effective modification strategies and can provide strong protection for various decompilers. However, the protection offered by MDN and CACoP is limited and requires further optimization. For MDN, which showed no effectiveness on current versions of decompilers, further exploration of new modification techniques or combining them with other modification strategies is necessary. CACoP, UUCM, and MOFS should also be optimized according to the performance of different decompilers to ensure they function well across various versions. Overall, adapting modification strategies to different tool versions and ensuring their compatibility with updates is the key direction for optimizing protection.

Overhead. To evaluate the performance impact of our file format mutation-based anti-decompilation techniques, we applied each strategy individually across a benchmark set of 100 Android applications. The average processing time per app per strategy was measured at 15.68 seconds. The actual mutation operations incurred negligible time costs; most of the overhead arose from standard APK procedures such as unpacking, repackaging, and signing. Compared to conventional protection mechanisms—such as code obfuscation, encryption, or runtime hardening—which often require several minutes to process a single application [31], [32], [33], our approach demonstrates a substantial improvement in efficiency.

In terms of storage and runtime overhead, the impact was minimal. As summarized in Table VII, five of the six strategies increased the app size by no more than 10 bytes, while only one (DBR) resulted in a slight increase on the order of kilobytes. Memory footprint during runtime remained unchanged in all cases except DBR, where a minor increase was observed due to JIT-related effects. On average, the unpacked APK size increased by less than 5%, confirming that our method introduces negligible overhead and remains suitable for large-scale deployment.

VI. Limitations

APKArmor is a security tool that enhances Android app protection through deep analysis and targeted modification strategies against decompilation. It processes installable app samples and significantly increases resistance to common decompilation tools, thereby safeguarding source code and core logic from reverse engineering by malicious users or competitors. Experimental results show that six of the seven strategies remain effective against the latest decompilation tools while preserving APK functionality, though some limitations persist.

Tool Dependencies. These protection methods mainly exploit the parsing mechanisms and vulnerabilities of mainstream decompilation tools (e.g., APKTool, JADX, Androguard). However, if these tools are updated or the vulnerabilities patched, their effectiveness may decline considerably. This indicates that maintaining long-term protection requires continuous adjustment and updating of strategies as decompilation tools evolve.

Cross-Platform Compatibility. In the experiments, all APKs were tested in a standard Android environment. However, the real Android ecosystem is highly diverse due to differences in devices and operating system versions. Some modification strategies may experience compatibility issues on specific devices or custom ROMs [34], leading to app installation failures or abnormal runtime behavior. Therefore, verifying the effectiveness of these methods across different hardware

and software environments will be an important direction for future research.

Potential Performance Impact. Although this paper aims to provide lightweight protection strategies, some methods (e.g., UUCM and DRA) may introduce extra decompression time or resource-loading delays, potentially degrading user experience. While these issues were not prominent in experiments, apps with higher performance demands (e.g., real-time video processing or high-load computing) may require further optimization to balance protection and performance.

Threat of Dynamic Analysis. The protection methods in this paper mainly target static analysis and decompilation attacks. However, with the development of dynamic analysis [35] tools such as Frida [36] and LSPosed [37], these tools can access an app's behavioral logic through runtime injection and memory analysis. Therefore, the current methods have limited ability to defend against dynamic analysis and need to be complemented with dynamic protection techniques to address the shortcomings of static protection.

Limitations in Application Scenarios. For apps with high-security requirements, such as those in finance and healthcare, the protection methods based on decompilation tool vulnerabilities may not meet strict compliance and security standards.

Threats from Advanced Attackers. Against highly skilled attackers, the current protection methods may not provide absolute security. Advanced attackers can manually analyze the APK structure, fix issues introduced by modification strategies, or develop custom tools to bypass these protective measures. Therefore, these methods are more suitable for delaying attacks rather than providing an absolute defense.

VII. Related Work

In this section, we review existing research on Android application protection against reverse engineering and decompilation. While prior work has extensively explored code-level protection mechanisms such as obfuscation, encryption, and hardening techniques, there has been limited investigation into file format-based approaches like our proposed file-based format adversarial techniques. This gap in research is particularly notable given the potential advantages of format-based protections in terms of efficiency and implementation simplicity compared to traditional methods.

Code obfuscation is one of the most widely studied methods to hinder understanding of decompiled code. Faruki et al. [38] reviewed Android code protection techniques, showing that while obfuscation complicates analysis, it rarely prevents sophisticated attackers from understanding the logic. Building on this, Ebad and Darem [39] examined obfuscation techniques and emphasized that reverse engineering tools are increasingly effective in defeating them. Dong et al. [40] reinforced this view through a large-scale study that identified and classified obfuscation strategies in the wild, particularly in malware, noting that obfuscation often provides only temporary protection as automated tools can reverse-engineer parts of the code. Kargén et al. [41] further analyzed obfuscation in both benign and malicious Android apps, showing that while obfuscation remains a critical defense, it is increasingly bypassed by advanced reverse engineering techniques.

Encryption and hardening techniques represent another significant branch of research in Android application protection. Kalauner [42] conducted a detailed analysis of Android anti-reverse engineering mechanisms and proposed ways to bypass encryption-based defenses, revealing that while encryption can effectively hide sensitive information, it adds computational overhead and can introduce new vulnerabilities if implemented incorrectly. Similarly, Kalysch [43] explored Android application hardening techniques, including encryption, that reduce the application's attack surface, though these approaches often came at the cost of degraded application performance and compatibility. In the realm of hardening techniques, methods such as code injection, instruction rearrangement, and virtualization have been proposed to thwart reverse engineering. Haupert et al. [44] examined the state of Android app hardening and demonstrated that while hardening significantly increases the difficulty of reverse engineering, it often introduces substantial trade-offs, including increased app size and slower performance. These findings were echoed by Sihag et al. [45] in their comprehensive review of Android malware and application hardening strategies, which emphasized the pressing need for lightweight and efficient methods to improve app security without significantly affecting usability.

More recently, format-based countermeasures have emerged as a promising direction, exploiting the flexibility of file formats such as ZIP (used in APKs) to hinder reverse engineering tools from processing APKs correctly. Badhani and Muttoo [46] showed that combining obfuscation and stealth techniques with file format manipulation can significantly strengthen Android applications against reverse engineering. Building on this trend, our paper explores new vulnerability-based techniques to disrupt decompilation, focusing on manipulating ZIP file headers and APK structures. We aim to provide a more efficient, low-cost solution than traditional obfuscation and encryption, while preserving application functionality and performance.

VIII. Conclusion

In this paper, we present a novel file format-based anti-decompilation strategy that effectively protects Android applications by exploiting structural vulnerabilities in decompilation tools. Our approach does not require code modifications and introduces minimal overhead (¡5% size increase), making it particularly suitable for resource-constrained developers. Through systematic evaluation on 100 production apps, we demonstrate that key techniques like pseudo-encryption and dirty bytecode replacement achieve 100% decompilation failure rates against state-of-the-art tools while maintaining full app functionality.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China (grants No. 62502168, 62572209), the Open Research Fund of the State Key Laboratory of Blockchain and Data Security (A2558), and the National Key Laboratory of Data Space Technology and System (QZQC2024006).

REFERENCES

[1] E. Protalinski, "Android passes 2.5 billion monthly active devices — venturebeat," Online, 2019, [Online]. Available: https://venturebeat.com/business/android-passes-2-5-billion-monthly-active-devices/.

[2] H. R. S, T. P, and S. Nandi, "Reverse engineering techniques for android systems: A systematic approach," in *2023 IEEE Guwahati Subsection Conference (GCON)*, 2023, pp. 1–6.

[3] V. C. . M. R. Team, "Navigating intellectual property rights in android app development," Online, 2024, [Online]. Available: https://moldstud.com/articles/p-navigating-intellectual-property-rights-in-android-app-development.

[4] M. R. Firm, "Mobile application security market size - by solution (anti-virus, anti-theft, web security, data backup & recovery), services (managed, professional), enterprise size (large enterprises, sme), deployment model, industry vertical & forecast, 2024 - 2032," 2023. [Online]. Available: https://www.gminsights.com/industry-analysis/mobile-application-security-market

[5] Techaisle. (2023) Global us$84b spend on it security in 2023 by smb and midmarket firms. Techaisle. Techaisle Blog. [Online]. Available: https://www.techaisle.com/blog/global-us84b-spend-on-it-security-2023

[6] Y. Zhuang, "The performance cost of software obfuscation for android applications," *Computers Security*, vol. 73, pp. 57–72, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404817302092

[7] skylot, "jadx," Online, 2024, [Online]. Available: https://github.com/skylot/jadx.

[8] iBotPeachest, "apktool," Online, 2024, [Online]. Available: https://github.com/iBotPeaches/Apktool.

[9] androguard, "androguard," Online, 2024, [Online]. Available: https://github.com/androguard/androguard.

[10] J. Xu, L. Zhang, D. Lin, and Y. Mao, "Recommendable schemes of anti-decompilation for android applications," in *2015 Ninth International Conference on Frontier of Computer Science and Technology*. IEEE, 2015, pp. 184–190.

[11] C. Collberg, "A taxonomy of obfuscating transformations," Technical Report 148, Tech. Rep., 1997.

[12] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *2010 International conference on broadband, wireless computing, communication and applications*. IEEE, 2010, pp. 297–300.

[13] Y. Zhao, Z. Tang, G. Ye, D. Peng, D. Fang, X. Chen, and Z. Wang, "Compile-time code virtualization for android applications," *Computers & Security*, vol. 94, p. 101821, 2020.

[14] A. Sarkar, A. Goyal, D. Hicks, D. Sarkar, and S. Hazra, "Android application development: a brief overview of android platforms and evolution of security systems," in *2019 Third International conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*. IEEE, 2019, pp. 73–79.

[15] B. Saikoa, "Dexguard," Online, [Accessed: 09-Apr-2020].

[16] L. Licel, "Stringer java obfuscator," Online, 2020, [Accessed: 09-Apr-2020]. Available: https://jfxstore.com/stringer/.

[17] ——, "Dexprotector–cutting edge obfuscator for android apps," Online, 2020, [Accessed: 09-Apr-2020]. Available: https://dexprotector.com/.

[18] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on android malware," *Computers Security*, vol. 51, pp. 16–31, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016740481500022X

[19] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998, pp. 184–196.

[20] M. D. Preda and F. Maggi, "Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology," *Journal of Computer Virology and Hacking Techniques*, vol. 13, pp. 209–232, 2017.

[21] Y. Piao, J.-H. Jung, and J. H. Yi, "Server-based code obfuscation scheme for apk tamper detection," *Security and Communication Networks*, vol. 9, no. 6, pp. 457–467, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.936

[22] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *Acm computing surveys (csur)*, vol. 49, no. 1, pp. 1–37, 2016.

[23] F. B. Cohen, "Operating system protection through program evolution," *Computers Security*, vol. 12, no. 6, pp. 565–584, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167404893900549

[24] Z. Kan, H. Wang, L. Wu, Y. Guo, and G. Xu, "Deobfuscating android native binary code," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 322–323.

[25] S. Exchange, "Information security stack exchange - information security qa," Online, [Accessed: 15-Jan-2025]. Available: https://security.stackexchange.com/.

[26] Reddit, "Reddit - netsec: Network security," Online, [Accessed: 15-Jan-2025]. Available: https://www.reddit.com/r/netsec/.

[27] XianZhi, "Xianzhi security community," Online, [Accessed: 15-Jan-2025]. Available: https://xz.aliyun.com/.

[28] KanXue, "Kanxue security forum," Online, [Accessed: 15-Jan-2025]. Available: https://www.kanxue.com/.

[29] I. GitHub, "Github - where the world builds software," Online, [Accessed: 15-Jan-2025]. Available: https://github.com/.

[30] L. Li, J. Gao, M. Hurier, P. Kong, T. F. Bissyandé, A. Bartel, J. Klein, and Y. L. Traon, "Androzoo++: Collecting millions of android apps and their metadata for the research community," *arXiv preprint arXiv:1709.05281*, 2017, [Accessed: 15-Jan-2025]. Available: https://arxiv.org/abs/1709.05281.

[31] O. Mirzaei, J. M. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano, "Androdet: An adaptive android obfuscation detector," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, 2019, pp. 301–312.

[32] Q. Zeng, L. Su, H. Zhang, M. Wan, and X. Luo, "Resilient dexguard: Hardening android application protection," in *Annual Computer Security Applications Conference*, ser. ACSAC '20, 2020, pp. 362–374.

[33] H. Wang, Y. Liu, L. Li, Y. Li, Y. Guo, G. Xu, Y. Liu, and X. Zhang, "Understanding and evaluating commercial android app hardening services," in *30th USENIX Security Symposium*, ser. USENIX Security '21, 2021, pp. 3247–3264.

[34] M. Gupta, A. Bhardwaj, and L. Garg, "Custom rom's in android'," *IJCSIT) International Journal of Computer Science and Information Technologies*, vol. 6, no. 2, pp. 1874–1875, 2015.

[35] A. Druffel and K. Heid, "Davinci: Android app analysis beyond frida via dynamic system call instrumentation," in *Applied Cryptography and Network Security Workshops: ACNS 2020 Satellite Workshops, AIBlock, AIHWS, AIoTS, Cloud S&P, SCI, SecMT, and SiMLA, Rome, Italy, October 19–22, 2020, Proceedings 18*. Springer, 2020, pp. 473–489.

[36] oleavr, "frida," Online, 2024, [Online]. Available: https://github.com/frida/frida.

[37] LSPosed, "Lsposed," Online, 2024, [Online]. Available: https://github.com/LSPosed/LSPosed.

[38] P. Faruki, H. Fereidooni, V. Laxmi, M. Conti, and M. Gaur, "Android code protection via obfuscation techniques: past, present and future directions," *arXiv preprint arXiv:1611.10231*, 2016.

[39] S. A. Ebad and A. A. Darem, "Exploring android obfuscators and deobfuscators: An empirical investigation," *Electronics*, vol. 13, no. 12, p. 2272, 2024.

[40] 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: 14th international conference, secureComm 2018, Singapore, Singapore, August 8-10, 2018, proceedings, part i*. Springer, 2018, pp. 172–192.

[41] U. Kargén, N. Mauthe, and N. Shahmehri, "Characterizing the use of code obfuscation in malicious and benign android apps," in *Proceedings*

of the 18th International Conference on Availability, Reliability and Security, 2023, pp. 1–12.

[42] P. G. Kalauner, "Analysis and bypass of android application anti-reverse engineering mechanisms," Ph.D. dissertation, Technische Universität Wien, 2023.

[43] A. Kalysch, *Android Application Hardening: Attack Surface Reduction and IP Protection Mechanisms*. Friedrich-Alexander-Universitaet Erlangen-Nuernberg (Germany), 2020.

[44] V. Haupert, D. Maier, N. Schneider, J. Kirsch, and T. Müller, "Honey, i shrunk your app security: The state of android app hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2018.

[45] V. Sihag, M. Vardhan, and P. Singh, "A survey of android application and malware hardening," *Computer Science Review*, vol. 39, p. 100365, 2021.

[46] S. Badhani and S. K. Muttoo, "Evaluating android malware detection system against obfuscation and stealth techniques," *International Journal of Digital Technologies*, vol. 2, no. II, 2023.