Deobfuscating Drone Android
Applications through Deep Learning

A

Project Report

submitted

in partial fulfilment

for the award of the degree of

Bachelor of Technology

in Department of Information Technology

Supervisor

Mr. Vinesh Kumar Jain

Assistant Professor

Submitted By:

Sparsh Bansal (20EEAIT057)

Rajat Mishra (20EEAIT043)

Harsh Yadav (20EEAIT017)

Department of Computer Science and Engineering

Engineering College Ajmer

May, 2024

Candidate's Declaration

I hereby declare that the work, which is being presented in the Project, entitled **"Deobfuscating Drone Android Applications through Deep Learning"** in partial fulfilment for the award of Degree of "*Bachelor of Technology*" in Department of **Information Technology,** Engineering College Ajmer, Bikaner Technical University is a record of my own investigations carried under the Guidance of **Mr. Vinesh Kumar Jain (Assistant Professor),** Department of Computer Science & Engineering, Engineering College Ajmer. I have not submitted the matter presented in this report anywhere for the award of any other Degree.

Sparsh Bansal (20EEAIT057)

Rajat Mishra (20EEAIT043)

Harsh Yadav (20EEAIT017)

B.Tech IV Year

(Information Technology)

Engineering College, Ajmer

अभियांत्रिकी महाविद्यालय, अजमेर
Engineering College, Ajmer
(An Autonomous Institution of **Government of Rajasthan**)
Badliya Chauraha, Shrinagar Road, Ajmer – 305025

CERTIFICATE

This is to certify that ***Sparsh Bansal, Rajat Mishra, Harsh Yadav*** of VIII Semester, Bachelor of Technology (Information Technology) 2020-24, has submitted the Project titled **"*Deobfuscating Drone Android Applications through Deep Learning*"** in partial fulfilment for the award of the degree of Bachelor of Technology under Bikaner Technical University, Bikaner (Rajasthan).

Date: 18 May, 2024

Mr. Vinesh Kumar Jain	**Dr. Jyoti Gajrani &**	**Dr. Jyoti Gajrani**
Supervisor	**Mr. S.N Tazi**	Head of Department
	Coordinator	CSE

ACKNOWLEDGMENT

I take this opportunity to express my gratitude to all those people who have been directly and indirectly with me during the competition of this Project.

I pay thank to **Mr. Vinesh Kumar Jain** who has given guidance and a light to me during this major project. His versatile knowledge about **"Deobfuscating Drone Android Applications through Deep Learning"** has eased me in the critical times during the span of this Project.

I acknowledge here out debt to those who contributed significantly to one or more steps. I take full responsibility for any remaining sins of omission and commission.

Sparsh Bansal (20EEAIT057)

Rajat Mishra (20EEAIT043)

Harsh Yadav (20EEAIT017)

B.Tech IV Year
(Information Technology)
Engineering College, Ajmer

ABSTRACT

In recent years, the proliferation of drones has significantly expanded their applications across various domains, from recreational photography to critical infrastructure surveillance. However, this rapid evolution has brought about new challenges, particularly in ensuring the security and integrity of drone software. One of the key strategies employed to protect drone applications from reverse engineering and tampering is obfuscation. Obfuscation techniques aim to obscure the codebase, making it difficult for adversaries to understand the underlying logic and extract sensitive information. While obfuscation serves as a fundamental defence mechanism, it also poses challenges for developers and security analysts attempting to analyse and verify the behaviour of drone applications. In response to the growing need for understanding the intricacies of obfuscated drone software, this report delves into the realm of deobfuscation and obfuscation techniques specific to Android-based drone applications. By unravelling the layers of obfuscation employed in these applications, we aim to shed light on the underlying functionalities and potential vulnerabilities , As utilization of obfuscation techniques in Android applications poses a significant challenge in identifying tampered code introduced by attackers or malware. As a consequence of obfuscation, the code becomes obscured, complicating the process of identifying potential risks or malicious modifications. Therefore, it becomes imperative to employ deobfuscation methods to revert the code to its original form, enabling thorough analysis and scrutiny for any nefarious alterations. This necessitates a meticulous approach to ensure the security and integrity of Android applications in the face of evolving threats. We intend to raise awareness about the critical role of deobfuscation in mitigating potential attacks on drone Android applications. By bridging the gap between theory and practice in drone software security, we aspire to foster a safer and more resilient ecosystem for the integration of drones across diverse applications. Through this endeavor, developers, security analysts, and policymakers can be better equipped to mitigate risks and ensure the integrity of drone technologies in contemporary society.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter-1

INTRODUCTION

In recent years, the proliferation of drones has significantly expanded their applications across various domains, from recreational photography to critical infrastructure surveillance. However, this rapid evolution has brought about new challenges, particularly in ensuring the security and integrity of drone software. One of the key strategies employed to protect drone applications from reverse engineering and tampering is obfuscation.

Obfuscation techniques aim to obscure the codebase, making it difficult for adversaries to understand the underlying logic and extract sensitive information. While obfuscation serves as a fundamental defence mechanism, it also poses challenges for developers and security analysts attempting to analyse and verify the behaviour of drone applications.

In response to the growing need for understanding the intricacies of obfuscated drone software, this report delves into the realm of deobfuscation and obfuscation techniques specific to Android-based drone applications. By unravelling the layers of obfuscation employed in these applications, we aim to shed light on the underlying functionalities and potential vulnerabilities.

This study is structured to provide insights into both sides of the coin: deobfuscation, which involves reversing the obfuscated code to its original form, and obfuscation, which explores strategies to enhance the protection of drone applications against malicious actors. Through empirical analysis and experimentation, we seek to identify the strengths and limitations of existing obfuscation techniques and propose novel approaches to bolster the security posture of Android drone applications.

By bridging the gap between theory and practice in the realm of drone software security, this report aims to equip developers, security analysts, and policymakers with valuable insights to mitigate the risks associated with the proliferation of drones in contemporary society. Ultimately, our endeavour is rooted in fostering a safer and more resilient ecosystem for the integration of drones across diverse applications.

1.1 Scope

This study delves into the realm of obfuscation and deobfuscation techniques tailored for Android-based drone applications. Its primary focus lies in unraveling the complexities of obfuscated drone software prevalent within the Android ecosystem. By dissecting the layers of obfuscation used in these applications, the study aims to reveal their original form, shedding light on their underlying functionalities and potential vulnerabilities. Through empirical analysis and experimentation, it seeks to practically assess the effectiveness of various obfuscation techniques, offering valuable insights into their strengths and limitations. Furthermore, the study extends its scope to identifying security risks embedded within obfuscated drone software. By unveiling security vulnerabilities and hidden malicious behaviors, developers and security analysts can effectively mitigate potential threats. Additionally, the study facilitates the validation and auditing of third-party libraries or dependencies integrated into Android applications, ensuring the integrity and security of external components.

1.2 Objectives

The objectives of this study revolve around achieving a comprehensive understanding and evaluation of obfuscation and deobfuscation techniques specific to Android-based drone applications. Its primary goal is to reverse-engineer obfuscated code to its original, human-readable form, thereby identifying vulnerabilities, weaknesses, and potential attack surfaces within obfuscated drone applications. Additionally, the study aims to evaluate the effectiveness of obfuscation techniques through empirical analysis, providing practical insights into their efficacy. Furthermore, it endeavors to enhance security measures deployed in Android-based drone applications based on insights gained from deobfuscation efforts. By understanding obfuscation strategies employed by malicious actors, developers can devise proactive security measures to safeguard sensitive information effectively. Ultimately, the study seeks to promote best practices in drone software development and security, empowering developers, security analysts, and policymakers with valuable insights to fortify the security posture of Android-based drone applications against evolving cyber threats

1.3 Obfuscation

Obfuscation in the context of software development, particularly in Android applications, involves the deliberate act of making code more difficult to understand or reverse engineer. This is typically achieved through techniques such as renaming variables and methods to obscure their original purpose, adding redundant or confusing code, or using encryption to obfuscate strings or sensitive information. The primary goal of obfuscation is to deter reverse engineering attempts by making it challenging for attackers to comprehend the inner workings of an application. Developers often employ obfuscation as a security measure to protect their intellectual property, prevent unauthorised access to proprietary algorithms or sensitive data, and mitigate the risk of piracy or unauthorised distribution.

1.4 Deobfuscation

Deobfuscation is the process of reversing the obfuscation applied to a piece of software in order to restore its original readability and functionality. In the context of Android applications, deobfuscation involves unravelling the obscured code to understand its logic and behaviour. This is typically done using various reverse engineering techniques, such as analysing bytecode, disassembling and debugging the application, or using specialised tools designed for deobfuscation purposes. Deobfuscation is often necessary for security researchers, analysts, or developers who need to analyse or modify an application for legitimate purposes, such as identifying security vulnerabilities, debugging issues, or enhancing the application's functionality.

1.5 Harms of Obfuscation and Publication in the Face of Attacks or Malware

While obfuscation can enhance the security of Android applications by making them more resistant to reverse engineering and tampering, it is not without its drawbacks. In some cases, overly aggressive obfuscation techniques can introduce performance overhead or compatibility issues, potentially impacting the user experience or functionality of the application. Moreover, obfuscation alone may not provide comprehensive protection against sophisticated attackers or malware, as determined adversaries can employ advanced reverse engineering techniques to

bypass obfuscation barriers. Additionally, the publication of obfuscated code without proper safeguards can inadvertently facilitate malicious activities, such as the creation of repackaged or counterfeit applications, injection of malware or backdoors, or exploitation of vulnerabilities for illicit purposes.

1.6 Utilisation of Deobfuscation to Mitigate Problems

Deobfuscation plays a crucial role in mitigating the challenges posed by obfuscated Android applications. Security researchers, analysts, and developers leverage deobfuscation techniques to analyse, understand, and potentially neutralise threats posed by obfuscated code. By deobfuscating Android applications, experts can identify security vulnerabilities, uncover malicious behaviours, and develop countermeasures to protect users and mitigate the risks associated with obfuscation. Deobfuscation also facilitates the auditing and validation of third-party libraries or dependencies used in Android applications, helping developers ensure the integrity and security of their software. Furthermore, the insights gained from deobfuscation can inform the development of more robust security solutions and best practices to fortify Android applications against evolving threats and attack vectors.

.

Chapter-2

LITERATURE SURVEY

2.1 Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations

The Android operating system, commanding an 87% market share, presents challenges for forensic investigators due to the rapid influx of new apps and updates (proposed by Chau and Reith). Obfuscation techniques in Android app development further compound these challenges, necessitating manual analysis and advanced tools (proposed by Wermke). Approximately 25% of Android apps employ obfuscation, rising to 50% among popular apps, with obfuscation also utilised by malware developers to evade detection . Investigative efforts focus on understanding encryption mechanisms and identifying malicious activity, underscoring the need for continual adaptation to evolving techniques (proposed by Zhang ; Wermke ; Zhou and Jiang,; Hammad). Obfuscation complicates reverse engineering processes, necessitating sophisticated methods to overcome these hurdles.

Tools for detecting and deobfuscating obfuscated code are essential for forensic analysis, yet many remain unreleased to the public, limiting their accessibility (proposed by Zhang). While obfuscation resilience aids in malware detection, it poses challenges for reverse engineering efforts, particularly with the possibility of employing multiple obfuscation methods sequentially (proposed by Pasetto; You and Yim; Mirzaei). This work presented a condensed summary of the status quo of Android obfuscation, obfuscation detection, and deobfuscation.

The obfuscation techniques have been separated by their utilised technique (e.g., identifier renaming or encryption), allowing examiners to easily locate them in this article and inform themselves. In contrast, obfuscation detection and deobfuscation have been organised by tools, as we believe sophisticated tools are most helpful from a practitioner's perspective. Consequently, this article offers a comprehensive overview suitable for beginners and intermediates alike.

"Android Application Forensics: A Survey of Obfuscation, Obfuscation Detection, and Deobfuscation Techniques and Their Impact on Investigations" provides a comprehensive exploration of the challenges and strategies inherent in navigating the complex landscape of Android app analysis. The abstract introduces the dual challenge faced by law enforcement and forensic investigators due to the proliferation of apps and the increasing use of obfuscation techniques, emphasizing the need for a thorough understanding of these techniques to effectively conduct investigations.

Table 2.1 Overview of obfuscation techniques

Obfuscation technique	PTT	MBE	OPT	IME
APK				
Aligning		x	x	low
Manifest transformation		x	x	low
Renaming				
Identifier renaming	x	x		low
Package renaming	x	x		low
Native library stripping	x	x		low
Control flow modification				
Code shrinking (or tree-shaking)		x	x	low
Call indirection	x	x	x	low
Junk Code insertion	x	x		low
Code reordering	x	x		low
Reflection	x	x		high
Evasion attacks		x		high
Encryption				
Data encryption	x	x		low
Asset file encryption	x	x		mid
Class encryption	x	x		high
Native code encryption	x	x		high

The introduction sets the stage by highlighting the exponential growth of Android app development and its implications for forensic analysis, underscoring the significance of comprehending obfuscation methods in investigative processes. It emphasizes the importance of identifying obfuscation techniques, irrespective of investigative motive, and acknowledges the necessity of manual analysis in many cases.

The subsequent sections elucidate Android basics and forensic fundamentals essential for understanding obfuscation and deobfuscation techniques, navigating through unpacking, disassembling, and repacking processes, underscoring the evolving challenges in Android application forensics.

The conclusion synthesizes the findings, offering a nuanced exploration of Android application forensics and its interplay with obfuscation techniques, empowering practitioners with insights and a robust framework to navigate the evolving landscape of digital investigations.

Table 2.2 Obfuscation detection tools

Tool	Obfuscation detection	Technique	Available
RepDroid	Repacking	dynamic	x
WLTDroid	Repacking	dynamic	x
IREA	Identifier Renaming	static	x
	Reflection		
	Data Encryption		
AndrODet	Identifier Renaming	static	✓
	Data Encryption		
	Control Flow Obfuscation		

The discussion on methodology commends the comprehensive search strategy and structured approach while suggesting documentation of search criteria and clarification of ambiguity handling. The analysis of obfuscation techniques applauds the comprehensive coverage and clear classification while recommending the inclusion of visual aids and consolidation of subsections. The examination of obfuscation detection appreciates the comprehensive overview and critical evaluation of tools while suggesting clarification of evaluation criteria and exploration of emerging trends. The review of deobfuscation techniques lauds the comprehensive coverage and critical evaluation of tools while recommending the incorporation of comparative analysis, exploration of emerging trends, and documentation of evaluation methodology.

2.2 Deguard: Statistical Deobfuscation of Android Applications

These studies introduce novel approaches to deobfuscating Android APKs through probabilistic learning from extensive code repositories, termed "Big Code". With a focus on reversing layout obfuscation, a prevalent technique that complicates comprehension by renaming critical program elements such as classes, packages, and methods, both papers formulate the deobfuscation challenge within probabilistic graphical models.

Leveraging tailored features and constraints to ensure semantic fidelity and predictive accuracy in the Android environment, they achieve precision and scalability through robust inference and learning algorithms. Implemented as the DeGuard tool, these methods effectively reverse layout obfuscation deployed by ProGuard and predict third-party libraries in obfuscated APKs. Additionally, DeGuard adeptly identifies and renames obfuscated program elements in Android malware.

Empirical validation underscores DeGuard's efficacy, demonstrating its ability to recover a substantial portion of obfuscated program element names and accurately forecast imported libraries. Both papers highlight the benefits of probabilistic learning from "Big Code" in addressing core security challenges, emphasizing the practicality and effectiveness of the proposed approaches in enhancing Android application security.

The document outlines a novel approach to deobfuscating Android APKs through probabilistic learning from extensive code repositories, referred to as "Big Code." The primary focus lies in reversing layout obfuscation, a prevalent technique that obfuscates key program elements, hindering comprehension of program functionality. By framing the layout deobfuscation problem within a probabilistic graphical model, the approach employs a diverse set of features and constraints tailored to the Android environment. Leveraging sophisticated inference and learning algorithms ensures high prediction accuracy and scalability.

The introduction contextualizes the significance of layout deobfuscation in Android security, highlighting the challenges and benefits associated with this endeavor. It emphasizes the reliance on probabilistic learning from Big Code to tackle complex security tasks effectively.

In the overview section, the document provides a concise explanation of ProGuard's role in obfuscating Android applications and outlines the fundamental steps of the DeGuard system. It elucidates the transition from obfuscated to deobfuscated code, emphasizing the importance of the dependency graph representation and the derivation of syntactic and semantic constraints.

The subsequent sections systematically delve into the technical aspects of the proposed approach, detailing the formalization of the layout deobfuscation problem, the derivation of naming constraints, and the probabilistic prediction process. The utilization of Conditional Random Fields (CRFs) to model dependencies between program elements is elucidated, along with the inference of likely names for obfuscated elements based on learned feature functions and constraints.

Overall, the document presents a comprehensive and innovative strategy for Android deobfuscation, leveraging probabilistic learning from large code repositories to overcome challenges posed by layout obfuscation. Its systematic problem formulation, coupled with practical insights from implementation and experimentation, positions it as a valuable contribution to the field of Android security.

2.3 Deoptfuscator: Defeating Advanced Control-Flow Obfuscation Using Android Runtime (ART)

The paper titled "Deoptfuscator: Defeating Advanced Control-Flow Obfuscation Using ART" introduces a novel approach to addressing control-flow obfuscation in Android apps, with a specific focus on advanced obfuscation patterns introduced by tools like DexGuard. It presents Deoptfuscator, a tool designed to detect and deobfuscate control-flow obfuscation in Android apps. The methodology behind Deoptfuscator involves identifying opaque variables, analyzing obfuscation patterns, and selectively removing obfuscated code to restore apps to their original state. The authors conducted experiments to evaluate the tool's effectiveness, comparing it with ReDex optimization and assessing its ability to restore app functionality while maintaining similarity to the original code.

Strengths of the paper include its innovative approach, systematic methodology, experimental validation, and flexibility in deobfuscation strategies. The approach fills a significant gap in Android app security by targeting advanced control-flow obfuscation techniques. The paper provides a detailed explanation of the methodology, enhancing transparency and reproducibility. Thorough experiments validate the effectiveness of Deoptfuscator, using real-world Android apps obfuscated with DexGuard. The tool offers flexibility in deobfuscation strategies, allowing users to adjust the threshold for identifying obfuscated classes based on the Obfuscated Bytecode Ratio (OBR).

Areas for improvement include the limitation of scope to control-flow obfuscation introduced by DexGuard, potentially limiting its applicability to apps obfuscated by other tools. Future research could explore methods to generalize the approach to address obfuscation patterns from multiple sources. Additionally, expanding the evaluation dataset to include a wider variety of apps and larger sample sizes could further validate its effectiveness across different scenarios and app types. Integration challenges with apps equipped with anti-tampering mechanisms may pose issues to the execution of deobfuscated code, suggesting a need for further exploration of strategies to overcome integration challenges with such protection mechanisms.

The Android Operating System commands a considerable portion of the smartphone market, yet faces challenges from the proliferation of malicious apps targeting sensitive data. To counter such threats, developers employ obfuscation techniques, while malware authors exploit similar methods to evade detection. In response, tools like Deoptfuscator have been developed to detect and deobfuscate control-flow obfuscated Android apps, ensuring the security and integrity of applications. Control-flow obfuscation, a key focus among various obfuscation techniques, poses challenges for traditional deobfuscation tools. Deoptfuscator emerges as the first tool capable of effectively detecting and deobfuscating high-level control-flow obfuscation patterns, preserving the original functionality of apps.

Its effectiveness is validated through rigorous evaluation, showcasing its ability to accurately deobfuscate apps while maintaining functionality. Beyond detection, Deoptfuscator's open-source availability promotes transparency and facilitates adoption by developers and researchers, contributing significantly to Android application security. Obfuscation, a widely used technique in software development, complicates program analysis while maintaining functionality, with various tools offering tailored obfuscation functionalities. Control-flow obfuscation complicates program analysis by altering code execution paths, but modern compilers like R8 possess optimization techniques to counter this.

Opaque predicates and variables play critical roles in effective control-flow obfuscation, with different obfuscation levels posing varying challenges for optimization tools. Deoptfuscator's systematic deobfuscation process involves several steps, from unpackaging obfuscated APKs to repackaging deobfuscated files, demonstrating its comprehensive approach to restoring original functionality.

2.4 Efficient Deobfuscation of Linear Mixed Boolean-Arithmetic Expressions

The review delves into the intricate realm of deobfuscating Mixed Boolean-Arithmetic (MBA) expressions, a common technique employed in code obfuscation. Focused primarily on linear MBAs, the paper introduces the SiMBA (Simple MBA Simplifier) algorithm, designed to enhance the efficiency, simplicity, and performance of existing deobfuscation tools like

MBA-Blast and MBA-Solver. At the outset, the authors elucidate the complexities associated with MBA transformations, emphasizing the fusion of logical and arithmetic operations.

Traditional SAT solvers and mathematical tools are deemed inadequate due to the mixed nature of MBAs. Building upon previous works, notably the contributions of Liu et al. (2019), the paper acknowledges the groundwork laid by MBA-Blast and MBA-Solver in deobfuscating linear MBAs.

SiMBA is positioned as an advancement over existing tools, promising to efficiently deobfuscate all linear MBAs while ensuring simplicity and competitive performance. The algorithm's methodology is detailed, involving the computation of signature vectors, transforming MBAs into linear combinations of base bitwise expressions, and optionally seeking simpler solutions. Key contributions highlighted in the review include the utilization of mathematical tools such as Theorem 1 and Corollary 2 for expressing linear MBAs as affine combinations of bitwise expressions, the introduction of Corollary 3 establishing equivalence based on signature vectors, and the development of the SiMBA algorithm for efficient deobfuscation.

Performance evaluations showcase SiMBA's effectiveness across various datasets, demonstrating competitive runtime and simplification capabilities compared to existing tools like MBA-Blast and MBA-Solver. Its versatility is underscored by its ability to handle arbitrary expressions without specific input structure requirements, making it a valuable asset in simplifying linear MBAs.Looking ahead, SiMBA's potential extension to handle polynomial and nonpolynomial MBAs is highlighted, suggesting its applicability in broader contexts beyond linear expressions.

Overall, the review provides a comprehensive analysis of the challenges and advancements in deobfuscating linear MBAs, with SiMBA emerging as a robust and efficient solution. Its simplicity, competitive performance, and potential for handling more complex expressions position it as a significant contribution to the field of code obfuscation and reverse engineering.

2.5 Katalina: an open-source Android string deobfuscator

The examination delves into the intricate landscape of Android malware deobfuscation, shedding light on the persistent challenges posed by sophisticated string obfuscation techniques. It articulates the existing gap in freely available deobfuscation tools, juxtaposing this scarcity against the backdrop of Katalina's innovative paradigm shift in Android malware deobfuscation, all without financial constraints. Katalina's methodology revolves around the execution of Android bytecode within a meticulously controlled environment, marking a departure from conventional approaches.

By leveraging this approach, Katalina automatizes the arduous task of unraveling obfuscated strings, offering invaluable insights into the concealed nuances of malware behavior and intent. Central to its efficacy is the meticulous emulation of Android bytecode instructions, a process that significantly mitigates the manual labor typically associated with malware analysis.

Furthermore, Katalina's proficiency in decoding obfuscated strings not only expedites the identification of malware patterns but also empowers analysts to craft robust detection rules, thereby fortifying defense mechanisms against evolving malware threats. In essence, Katalina emerges as a beacon of promise for malware analysts, heralding a new era of streamlined deobfuscation processes and actionable intelligence in the ongoing battle against malware proliferation.

2.6 Macneto: Deobfuscating Android Applications through Deep Learning

Android apps are often obfuscated before distribution to reduce file size and deter unauthorised use. However, this practice also provides cover for malware, as obfuscated code can disguise malicious intent. Automated deobfuscators are in demand to uncover the original structure of obfuscated code. These tools serve various purposes, such as aiding plagiarism detection, identifying vulnerable third-party libraries, and assisting in code searches. Additionally, they support human-guided analysis for assessing security risks in applications. Deobfuscators typically use a training set of non-obfuscated code to create a model for deciphering obfuscated

code. Once trained, they can restore original method and variable names, and even reconstruct obfuscated method structures and code.

Some deobfuscation tools utilise the app's control flow graph to aid in this process. However, we find out they are susceptible to obfuscators that introduce extra basic blocks and jumps to the app’s code and can be slow to use, requiring many pairwise comparisons to perform their task from, F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, T. Jebara, S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla works and research. Using another approach called DeGuard, B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev tell us that Deguard is an advanced deobfuscator that constructs a probabilistic model for identifiers by analysing their co-occurrences within code (e.g., identifying relationships between variables, classes, and methods). While this method is fast once the statistical model is established, it struggles with obfuscators that alter the code layout, such as moving methods to different classes or introducing new fields.

We introduce MACNETO, a new approach to automated deobfuscation of Android apps that utilises recurrent neural networks and deep learning. MACNETO exploits a key insight about obfuscation: while obfuscators aim to drastically change code appearance, they retain the original semantics. By employing topic modelling, MACNETO learns the deep semantics of code behaviour, which remain stable despite obfuscation. These topic models, trained on a simple feature set representing bytecode instructions and Android APIs, are resilient to various obfuscation techniques, including identifier renaming, method call injection, method splitting, and control flow modifications.

MACNETO utilises offline deep learning to train a topic classifier on both obfuscated and unobfuscated apps, enabling adaptation to various obfuscators with new datasets for fast online deobfuscation. Real-world evaluation demonstrates significantly higher precision than statistical approaches, with 96% precision on ProGuard-obfuscated code and 91% precision on Allatori-obfuscated code. Despite using small training sets, MACNETO effectively obfuscate method names in malware and plagiarism-related tasks by exploiting semantic coherence.

This sophisticated strategy involves meticulously analysing instruction distributions, identifying crucial machine topics, and employing a powerful Recurrent Neural Network (RNN) to establish mappings between original and obfuscated methods, showcasing its unwavering effectiveness in offline scenarios. Deobfuscation is essential for reverting obfuscated programs to their original versions, aiding developers in comprehension and analysis.

MACNETO utilises topic modelling and deep learning for automatic deobfuscation. In experiments with 1600+ Android APKs obfuscated by ProGuard and Allatori, MACNETO achieves over 96% precision in recovering ProGuard-obfuscated programs and maintains high precision with Allatori. Compared to a basic approach, MACNETO shows up to a 200% improvement in deobfuscation precision.The literature review for "VisionWalk: Augmented Reality Campus Guide using Unity AR Foundation" explores the existing research and developments in augmented reality (AR) technologies, particularly in the context of campus navigation systems. Augmented reality has garnered significant attention due to its potential to overlay digital information onto the physical world, enhancing users' perception and interaction with their surroundings. Previous studies have investigated various aspects of AR applications, including user interface design, spatial mapping, and real-time data integration.

Additionally, the review examines prior work in the field of AR campus guides, highlighting notable projects and their contributions to enhancing campus navigation and information dissemination. These studies have explored different approaches to AR-based navigation, such as marker-based tracking, GPS localization, and computer vision techniques. Moreover, the literature review delves into the capabilities and features of Unity AR Foundation, a popular development framework for creating AR experiences across different platforms.

By synthesizing existing research and developments, the literature review establishes a foundation for the "VisionWalk" project, identifying gaps, challenges, and opportunities for innovation in the design and implementation of an AR campus guide using Unity AR Foundation. This comprehensive understanding of the state-of-the-art in AR technology and campus navigation systems informs the methodology and design decisions of the project, facilitating the development of an effective and user-friendly solution tailored to the needs of the target audience.

Chapter-3

DATASET ARCHITECTURE

This report outlines the process of gathering a dataset consisting of 60 drone applications, sourced from a variety of providers, including DJI (Dà-Jiāng Innovations Science and Technology Co., Ltd.) and third-party developers. The aim of this dataset is to conduct a comprehensive analysis on these applications, focusing on functionality, security, usability, and performance.

The process of gathering the dataset involved several steps:

- Research and Selection: Initially, extensive research was conducted to identify a diverse range of drone applications available in the market. This involved exploring app stores, developer websites, industry forums, and other relevant sources to compile a comprehensive list of potential candidates.

- Criteria Definition: Clear criteria were established to guide the selection process. These criteria included factors such as popularity, user ratings, feature richness, availability of APIs or SDKs, and compatibility with a variety of drone models.

- Data Collection: Once the list of potential applications was compiled, efforts were made to gather detailed information about each application. This included obtaining metadata such as application name, developer information, version history, release dates, and user reviews. Additionally, access to application binaries or APIs was sought where possible to facilitate deeper analysis.

- Dataset Curation: Finally, the dataset was curated to include a balanced selection of applications, representing various categories such as aerial photography, mapping and surveying, agricultural drones, industrial inspections, and recreational use.

The dataset comprises a total of 60 applications, meticulously gathered and vetted to ensure a comprehensive representation of the landscape.

- 3D survey pilot.apk
- aeroGCS green.apk
- air control.apk
- AirHub.apk
- Dji fly.apk
- Dji go.apk
- Dji smartfarm.apk
- Dmo pilot.apk
- Drone buddy.apk
- Drone surveyor.apk
- Dronecast.apk
- Dronedeploy.apk
- Dronelink.apk
- Droniq.apk
- PIX4Dcapture pro.apk
- Plex pilot.apk
- Rainbow.apk
- Smart flight.apk
- Uav forecast.apk
- Airhub-smartcontroller-660
- Android_vision_guanwang
- App-arm
- airhub-sc.apk
- Aptoide.apk
- AR.freeflight
- Drone App.apk
- DroneDepoly.apk
- Dronelink.apk
- DRONI.apk
- drony.apk
- flyingdronecamera.apk
- GoFlyD.J.IDrone.apk
- gosky.apk
- Litchi.apk
- Mi_Drone.apk
- P2Braz.apk
- SYMA-FPV.apk
- Vision+s11.apk
- Wifi-UAV.apk
- Tello.apk
- Shox-chorma.apk
- Ronin.apk
- quadcopter-drone-rc-all-drones-3-1.apk
- PrecisionFlight-2.0.apk
- Mi_Drone_vV_1.3.11.20601_www.9apps.com_.apk
- Luma-AI-v2.1.1.apk
- io-kittyhawk-1.apk
- hover-drone-and-uav-pilot-app.apk
- Fyd-uav.apk
- ELF_VRDrone-5.28.apk
- drone-harmony-for-dji-drones.apk
- djipilot-pe.apk
- DJI-Mimo.apk
- DJI-Go.apk

Chapter-4

METHODOLOGY

4.1 Methodology for Deobfuscation of Drone Android Application:

1. **Understanding the Obfuscation Techniques Used:**
    - Analyze the specific obfuscation techniques commonly employed in drone Android applications. This may include techniques such as ProGuard obfuscation, DexGuard control-flow obfuscation, and string encryption.
2. **Gathering Sample Drone Android Applications:**
    - Collect a diverse set of sample drone Android applications that are known to be obfuscated. These applications should represent various functionalities and complexities typical in drone applications.
3. **Preprocessing and Preparation:**
    - Extract the bytecode (DEX or APK) from the collected drone Android applications.
    - Perform preprocessing steps such as decompilation to obtain readable Java code from the bytecode.
4. **Selection of Deobfuscation Tools and Techniques:**
    - Evaluate and select appropriate deobfuscation tools and techniques based on the identified obfuscation methods and the nature of the drone applications.
    - Consider using tools such as DeGuard, Deoptfuscator, SiMBA, Katalina, or MACNETO, depending on the specific requirements and challenges posed by the obfuscated code.
5. **Deobfuscation Process:**
    - Apply selected deobfuscation tools and techniques to the preprocessed code to reverse the obfuscation and restore the original structure and functionality.
    - Utilize tools' functionalities to identify and reverse obfuscated program elements, libraries, and sensitive data usage within the drone applications.

6. **Validation and Verification:**
    - Validate the effectiveness of the deobfuscation process by comparing the deobfuscated code with the original, unobfuscated code where available.
    - Verify that the deobfuscated code accurately represents the intended functionality of the drone application and that critical components such as flight control algorithms and communication protocols remain intact and functional.
7. **Testing and Analysis:**
    - Conduct extensive testing and analysis of the deobfuscated drone Android applications to ensure their reliability, security, and compliance with regulatory standards.
    - Test the applications under various scenarios and conditions to verify their performance, stability, and safety.
8. **Documentation and Reporting:**
    - Document the deobfuscation process, including the tools and techniques used, as well as any challenges encountered and their resolutions.
    - Prepare comprehensive reports detailing the findings of the deobfuscation process, including insights gained, vulnerabilities discovered, and recommendations for improvement.
9. **Iterative Improvement:**
    - Iterate on the deobfuscation process based on feedback, lessons learned, and advancements in deobfuscation techniques and tools.
    - Continuously refine and enhance the methodology to address emerging obfuscation challenges and improve the effectiveness and efficiency of the deobfuscation process.

By following this methodology, developers, analysts, and security researchers can effectively deobfuscate drone Android applications, uncovering their underlying functionality, identifying potential security vulnerabilities, and ensuring their safety and reliability in various operational contexts.

4.2 The execution methods employed by each deobfuscator are as follows:

4.2.1 Deguard

- The obfuscation process is facilitated through the online platform available at http://apk-deguard.com/, offering a seamless experience by enabling users to upload APK files directly.
- This platform harnesses the power of DeGuard, a sophisticated technology built upon powerful probabilistic graphical models.
- Developed as part of the Machine Learning for Programming project at ETH Zurich, DeGuard specializes in extracting vital information from Android APKs, including method and class names, along with the identification of third-party libraries.

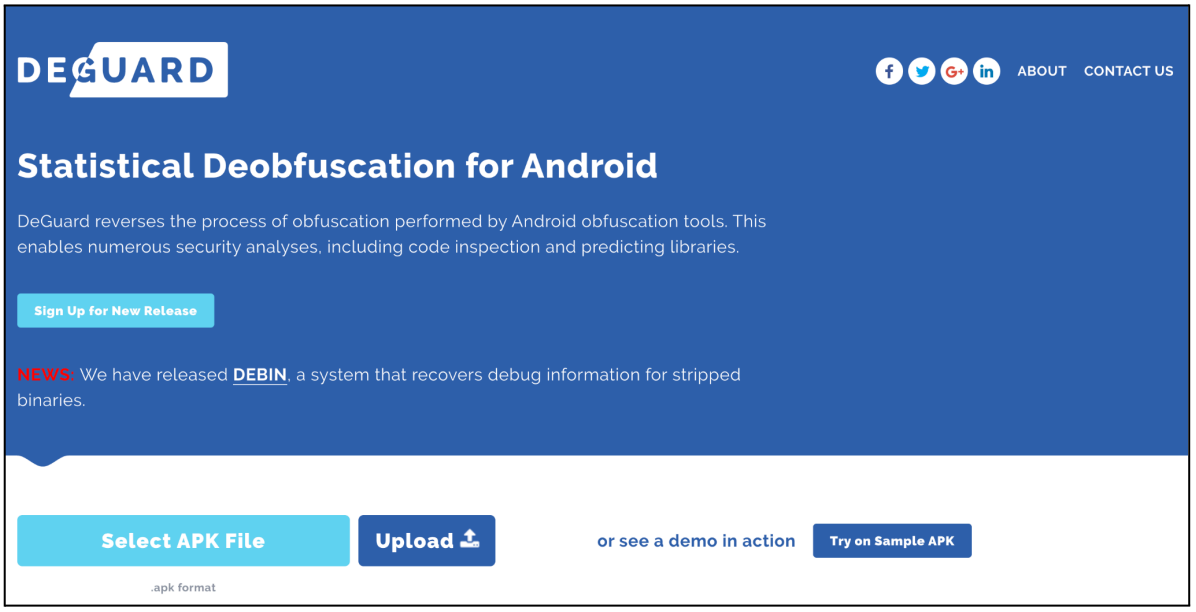


Fig 4.1 DeGuard Website

- Notably, DeGuard excels in revealing string decoders and classes responsible for handling sensitive data within Android malware.
- DeGuard's technology is underpinned by the open-source Nice2Predict framework (https://github.com/eth-sri/Nice2Predict), which provides a solid foundation in machine learning and programming, ensuring the reliability and effectiveness of the process.

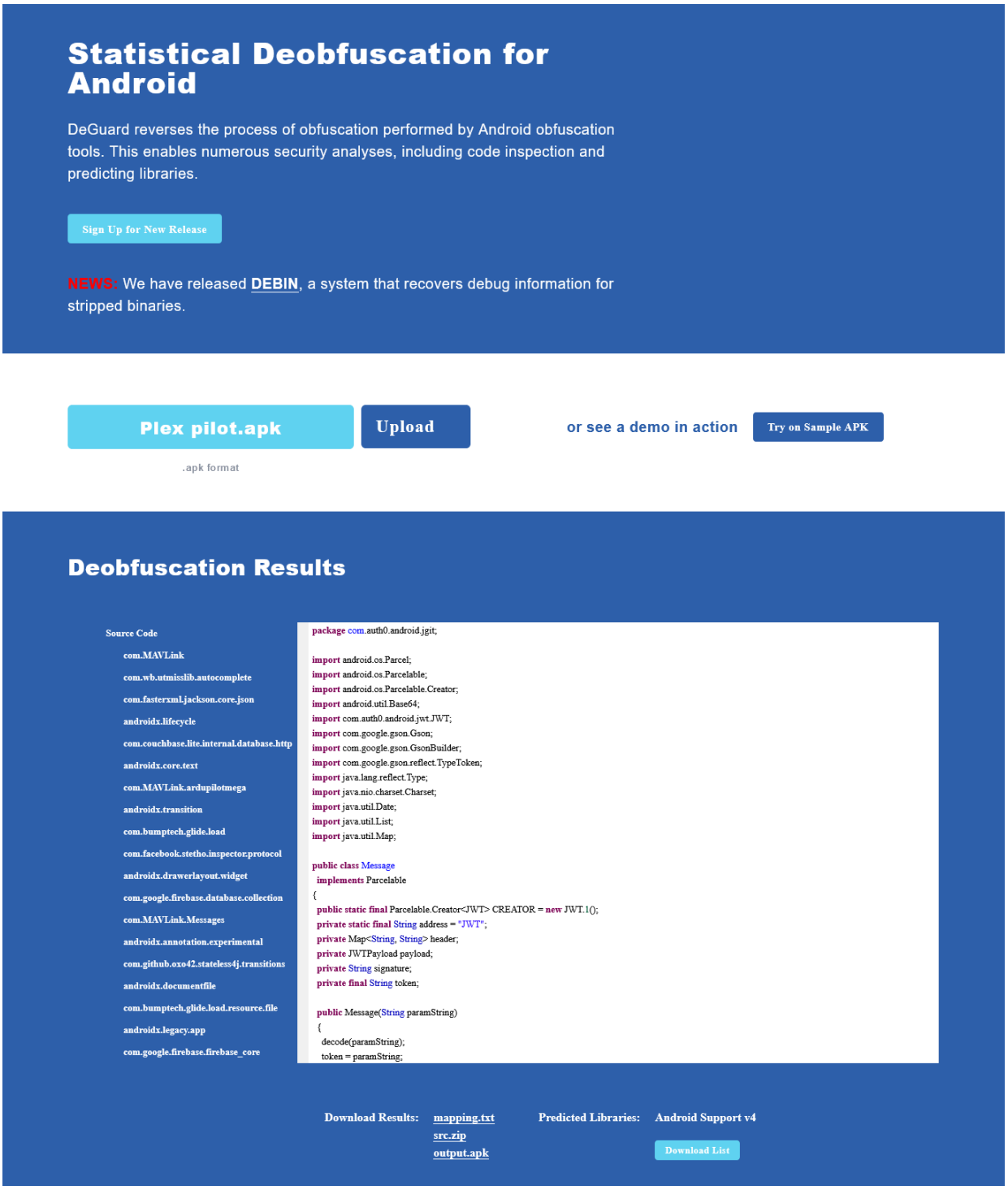
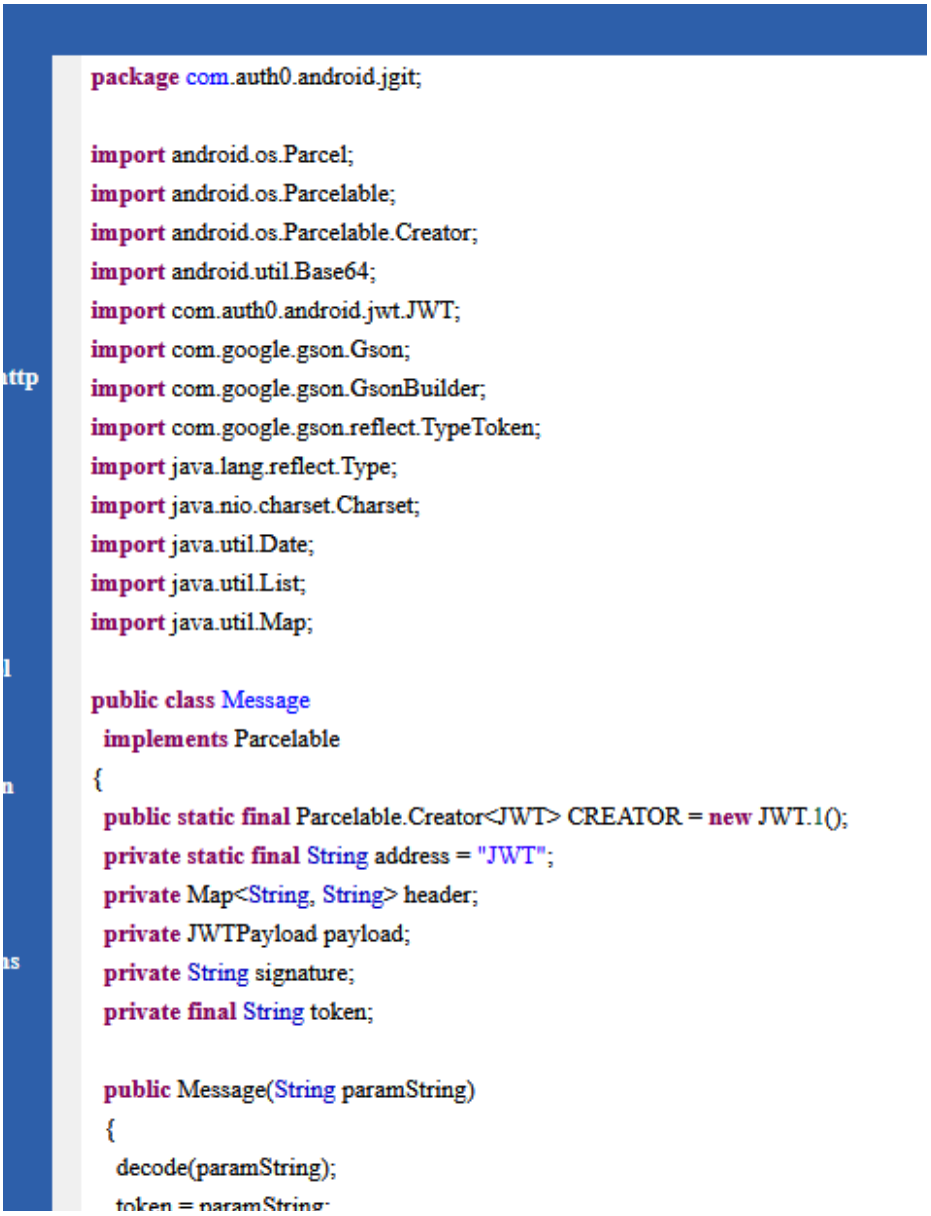


Fig 4.2 DeGuard Website result



```
package com.auth0.android.jgit;

import android.os.Parcel;
import android.os.Parcelable;
import android.os.Parcelable.Creator;
import android.util.Base64;
import com.auth0.android.jwt.JWT;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.reflect.TypeToken;
import java.lang.reflect.Type;
import java.nio.charset.Charset;
import java.util.Date;
import java.util.List;
import java.util.Map;

public class Message
 implements Parcelable
{
 public static final Parcelable.Creator<JWT> CREATOR = new JWT.1();
 private static final String address = "JWT";
 private Map<String, String> header;
 private JWTPayload payload;
 private String signature;
 private final String token;

 public Message(String paramString)
 {
  decode(paramString);
  token = paramString;
```

Fig 4.3 DeGuard Website Code Result

- The outcome presented here illustrates the outcome of the deobfuscation process conducted by DeGuard. It is evident that the identifiers have been restored to their original names prior to obfuscation, demonstrating the reversal of numerous obfuscation techniques employed.

4.2.2 Deoptfuscator

Prerequisites

In order to build and run deoptfuscator, the followings are required:

- Deoptfuscator based on Ubuntu 18.04 LTS 64bit PC

libboost, libjson (C++ library)

```
$ sudo apt-get install libboost-all-dev
$ sudo apt-get install libjsoncpp-dev
```

openjdk

```
$ sudo apt-get install openjdk-11-jdk
```

i386 libc

```
$ sudo dpkg --add-architecture i386
$ sudo apt-get update
$ sudo apt-get install libc6:i386 libstdc++6:i386
```

zipalign

```
$ sudo apt-get install zipalign
```

apksigner

```
$ sudo apt-get install apksigner
```

How to Install

- deoptfuscator's repositary need git-lfs
- Git LFS

```
$ curl -s https://packagecloud.io/install/repositories/github/git-lfs/script.deb.sh | sudo bash
$ sudo apt install git-lfs
$ git clone https://github.com/Gyoonus/deoptfuscator.git
```

```

```

Repository Already Contains Tools needed to run the Tool

- Apktools : https://ibotpeaches.github.io/Apktool/
- fbredex : https://fbredex.com/

HOW TO USE

Set Local Environment

```
$ . ./launch.sh
```

OR

```
 $ source ./launch.sh
```

Deobfuscate an Android application that has been transformed using control-flow obfuscated techniques.

```
$ python3 deoptfuscator.py <obfuscated_apk>
```

Then run this

```
$ python3 deoptfuscator.py
test/AndroZoo_DexGuard_apk/com.alienguns.scifirifles_4F326C99558145BB
636D31C96488823A.apk
```

- If the input file (an obfuscated app) was com.alienguns.scifirifles_4F326C99558145BB636D31C96488823A.apk, the file name of the deobfuscated apk is com.alienguns.scifirifles_4F326C99558145BB636D31C96488823A_deobfuscated_align.apk

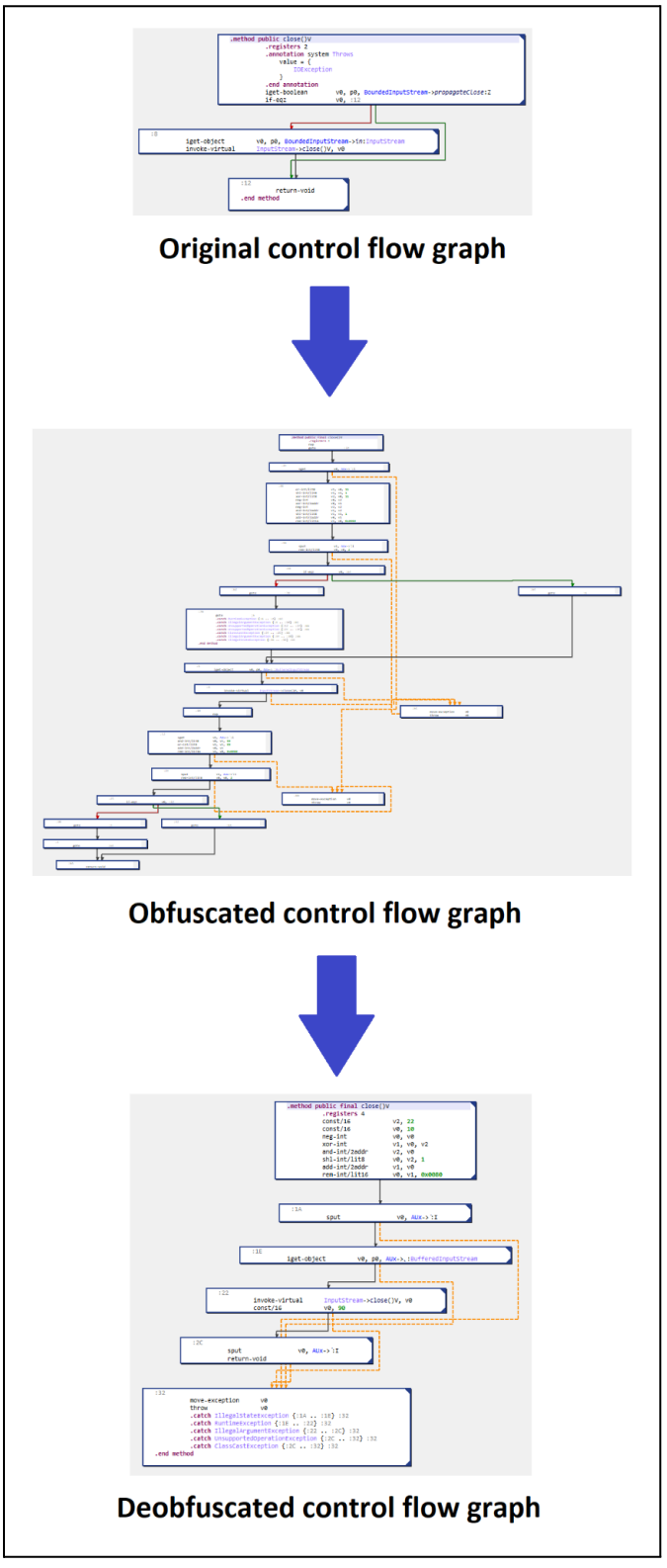


Fig 4.4 Deoptfuscator Handling Control-Flow Obfuscation

This tool demonstrates exceptional proficiency in deobfuscating Android applications subjected to the control flow obfuscation option of DexGuard, showcasing its robustness in reversing complex obfuscation patterns.

Presently, this tool is specifically tailored to address the control-flow obfuscation methodologies employed by DexGuard, offering a specialized solution for developers and security analysts grappling with this particular challenge.

It is essential to note that while this tool excels in handling control-flow obfuscation techniques utilized by DexGuard, its scope is currently limited to this specific aspect of obfuscation.

This tool, while highly effective in deciphering control-flow obfuscation, does not extend its capabilities to other prevalent obfuscation techniques such as layout obfuscation, identifier renaming, and string encryption.

4.2.3 SiMBA

SiMBASiMBA is a tool for the simplification of linear mixed Boolean-arithmetic expressions (MBAs). Like MBA-Blast and MBA-Solver, it uses a fully algebraic approach based on the idea that a linear MBA is fully determined by its values on the set of zeros and ones, but leveraging the new insights that a transformation to the 1-bit-space is not necessary for this.

Content

Two main programs (Python 3) are provided:

- simplify.py for the simplification of single linear MBAs
- simplify_dataset.py for the simplification of a set of linear MBAs contained in a file and their verification via a comparison with corresponding simpler expressions also contained in this file

Additionally, the program check_linear_mba.py can be used for checking whether expressions represent linear MBAs.

Usage

Simplifying single expressions

In order to simplify a single expression expr, use

```
python3 src/simplify.py "expr"
```

Alternatively, multiple expressions can be simplified at once, e.g.:

```
python3 src/simplify.py "x+x" "a&a"
```

In fact, each command line argument which is not an option is considered as an expression to be simplified. Note that omitting the quotation marks may imply undesired behavior. The simplification results are printed to the command line as shown in the following:

```
*** Expression x+x
*** ... simplified to 2*x
*** Expression a
*** ... simplified to a
```

Per default no check whether the input expression is a linear MBA is performed. This check can optionally be enabled via the option -l:

```
python3 src/simplify.py "x*x" -l
```

Since $x*x$ is no linear MBA, the following output would show up in this case:

```
*** Expression x*x
Error: Input expression may be no linear MBA: x*x
```

If option -z is used, the simplification results are finally verified to be equal to the original expressions using *Z3*. This does not effect the command line output as long as the algorithm works correctly and the input expression is a linear MBA:

```
python3 src/simplify.py "x*x" -z
```

This would trigger the following error:

```
*** Expression x*x
Error in simplification! Simplified expression is not equivalent to original one!
```

Since the constants occuring in SiMBA's output expressions are always nonnegative, they may depend on the number of bits used for constants as well as variables. This number is 64 by default and can be set using the option -b:

```
python3 src/simplify.py "-x" -b 32
```

For a number b of bits, the constants occuring in the output always lie between 0 and $2b-1$. Hence the call above would imply the following output:

```
*** Expression -x
*** ... simplified to 4294967295*x
```

Simplifying and verifying expressions from a file

In order to simplify expressions stored in a file with path path_to_file, use

```
python3 src/simplify_dataset.py -f path_to_file
```

That is, the file has to be specified using the option -f. Each line of the file has to contain a complex expression as well as an equivalent simpler one, separated by a comma, e.g.:

example-expressions.txt:

```
(x&y)+(x|y), x+y
(x|y)-(~x&y)-(x&~y), x&y
-(a|~b)+(~b)+(a&~b)+b, a^b
2*(s&~t)+2*(s^t)-(s|t)+2*~(s^t)-~t-~(s&t), s
```

For each line, both the complex and the simple expression are simplified and finally compared. The reason for simplifying the latter is to make verification results independent of whitespace, the order of factors or summands, etc.

As with simplify.py, a linearity check as well as a check for a correct simplification can be enabled using the options -l and -z, resp., and the number of bits can be specified using option -b. If one wants to run SiMBA on only a certain maximum number of expressions contained in the specified file, this maximum number can be specified via option -r:

```
python3 src/simplify_dataset.py -f some_file.txt -r 2
```

If some_file.txt would contain the expressions listed above, only the first two of them would be simplified:

```
Simplify expressions from data/some_file.txt ...
  * total count: 2
  * verified: 2
  * equal: 2
  * average duration: 0.00014788552653044462
```

In any case, the output gives information about

- the total number of expressions in the input,
- the number of expressions which could be verified to be equivalent to the corresponding simpler expression using Z3 after simplification (unless the simplification result already has the exact same string representation),

- the number of expressions which are simplified to the very same expression as the corresponding simple expression, and
- the average runtime in seconds.

Please note that an optional verification of a correct simplification using Z3 contributes to the runtime, while this is not the case for the comparison of the simplification results of the pairs consisting of a complex and a simpler expression.

Per default the simplification results are not printed, but only these statistics are presented. If information about the former is desired, the option -v can be used:

```
python3 src/simplify_dataset.py -f some_file.txt -v
```

The following output would then be shown:

```
Simplify expressions from data/some_file.txt ...

   *** 1 groundtruth x+y, simplified x+y => equal: True, verified: True
   *** 2 groundtruth x&y, simplified x&y => equal: True, verified: True
   *** 3 groundtruth a^b, simplified a^b => equal: True, verified: True
   *** 4 groundtruth s, simplified s => equal: True, verified: True

  * total count: 4
  * verified: 4
  * equal: 4
  * average duration: 0.00016793253598734736
```

Another option -e provides the possibility to encode all expressions' outputs by affine functions $f(x)=ax+b$ with random integers a,b between 1 and $2b-1$ if b is the number of bits:

```
python3 src/simplify_dataset.py -f some_file.txt -v -e
```

Of course the same function is applied to a pair of expressions in the same line. This would give output similar to the following:

```
Simplify expressions from data/some_file.txt ...

  *** 1 groundtruth
10623056950310032687+5038261596809828791*x+5038261596809828791*y, simplified
10623056950310032687+5038261596809828791*x+5038261596809828791*y => equal:
True, verified: True

  *** 2 groundtruth 15181401701264988765+39628685921311 93124*(x&y), simplified
1518140170126 4988765 +3962868592131 193124*(x&y) = > equal: True, verified: True

  *** 3 groundtruth 681244 0940417974076+11894131080657788315*(a^b), simplified
6812440940417974076 +11894131080657788315*(a^b) => equal: True, verified: True

  *** 4 groundtruth 4558303267887122851+10271005790757592209*s, simplified
4558303267887122851 +10271005790757592209*s => equal: True, verified: True

  * total count: 4
  * verified: 4
  * equal: 4
  * average duration: 0.00019435951253399253
```

Reproducibility

For a reproduction of part of the experiments stated in the paper, one may use any of the dataset files contained in the directory data/. For each of the following functions $e1,\ldots,e5$, datasets of 1,000 equivalent linear MBAs using 2, 3 or 4 variables are provided:

- $e1(x,y)=x+y$
- $e2=49{,}374$
- $e3(x)=3{,}735{,}936{,}685{,}x+49{,}374$
- $e4(x,y)=3{,}735{,}936{,}685{,}(x \wedge y)+49{,}374$
- $e5(x)=3{,}735{,}936{,}685 \cdot \sim x$

For $e1$, additional datasets for 5 to 7 variables are provided. These MBAs have been generated using an algorithm based on the method described by Zhou et al. in 2007 and described in the paper.

Please note that these datasets have been generated for $b=64$ bits. For different numbers of bits, their equivalence to the ei's cannot be guaranteed.

For the reproduction of further experiments, we refer to the datasets provided by the MBA-Solver repository and the NeuReduce repository, resp.

Checking linearity

The file check_linear_mba.py is used by the simplifier, but it also provides its own interface, e.g.:

```
python3 src/check_linear_mba.py "x+x" "x*x"
```

It checks all expressions which are passed via command line arguments. In this case it would imply the following output:

```
*** Expression x+x
*** +++ valid
*** Expression x*x
*** --- not valid
```

Format of MBAs

The number of variables is in theory unbounded, but of course the runtime increases with variable count. There is no strong restriction on the notation of variables. They have to start with a letter and can contain letters, numbers and underscores. E.g., the following variable names would all be fine:

- *a*, *b*, *c*, ..., *x*, *y*, *z*, ...
- *v*0, *v*1, *v*2, …
- *v*0,*v*1, *v*2, ..
- *X*0, *X*1, *X*2, ...
- *var*0, *var*1, *var*2, ...
- *var*1*a*, *var*1*b*, *var*1*c*, ...

The following operators are supported, ordered by their precedence in Python:

- ~, −: bitwise negation and unary minus
- *: product
- +, −: sum and difference
- &: conjunction
- ∧: exclusive disjunction
- |: inclusive disjunction

Whitespace can be used in the input expressions. E.g., the expression "x+y" may alternatively be written "x + y".

Please respect the precedence of operators and use parentheses if necessary! E.g., the expressions 1+(*x*|*y*) and 1+*x*|*y* are not equivalent since + has a higher precedence than |. Note that the latter is not even a linear MBA.

Dependencies

The SMT Solver Z3 is required

- by simplify_dataset.py where simplified expressions are verified to be equivalent to corresponding simple expressions, and
- by simplify.py if the optional verification of simplified expressions is used. If this option is unused, no error is thrown even if Z3 is not installed.

Installing Z3:

- from the Github repository: https://github.com/Z3Prover/z3, or
- on Debian: sudo apt-get install python3-z3

4.2.4 Katalina

The deobfuscation process is executed by initially converting the Android application package (APK) file into a Dalvik Executable (DEX) file format. Subsequently, the DEX file is inputted into Katalina, a tool designed for deobfuscating Android applications, which is openly

accessible on GitHub at the following repository: [Katalina GitHub Repository](https://github.com/huuck/Katalina?tab=readme-ov-file)

.

Initially, the process begins by obtaining the APK file slated for deobfuscation. Subsequently, the focus shifts towards locating the necessary DEX file within the application. To achieve this, the APK file undergoes compression into a ZIP archive, facilitating the extraction of the requisite DEX file essential for the deobfuscation procedure.

Katalina is like Unicorn but for Dalvik bytecode. It provides an environment that can execute Android bytecode one instruction at a time. Requires Python >= 3.10

How to install:

```
pip install -r requirements.txt
```

How to run:

```
python3 main.py -xe classes.dex
```

Katalina generates logs by parsing the bytecode extracted from the Dalvik executable file of the application. These logs offer insights into the behavior of the potentially contaminated or malicious code within a controlled environment. Analyzing these logs enables the observation of the actions performed by the infected code, facilitating a safer assessment of its functionality.

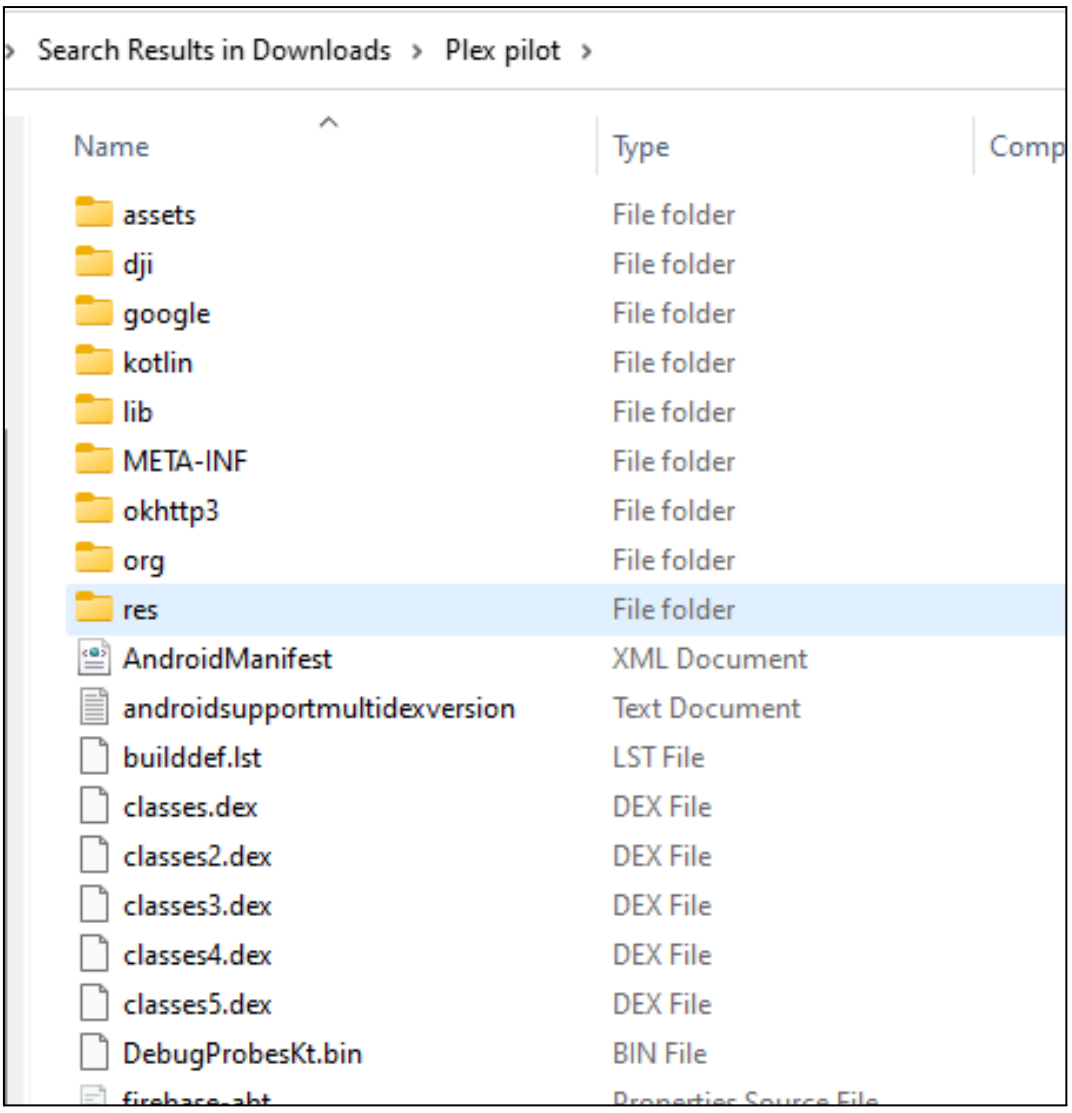


Fig 4.5 The extraction of the zipped application to obtain the DEX file.

This file comprises all the classes.dex files extracted from the application's compressed archive, essential for further processing.

```
python3 main.py -xe classes.dex
```

Executing this command yields output logs that demonstrate the functionality of the code.

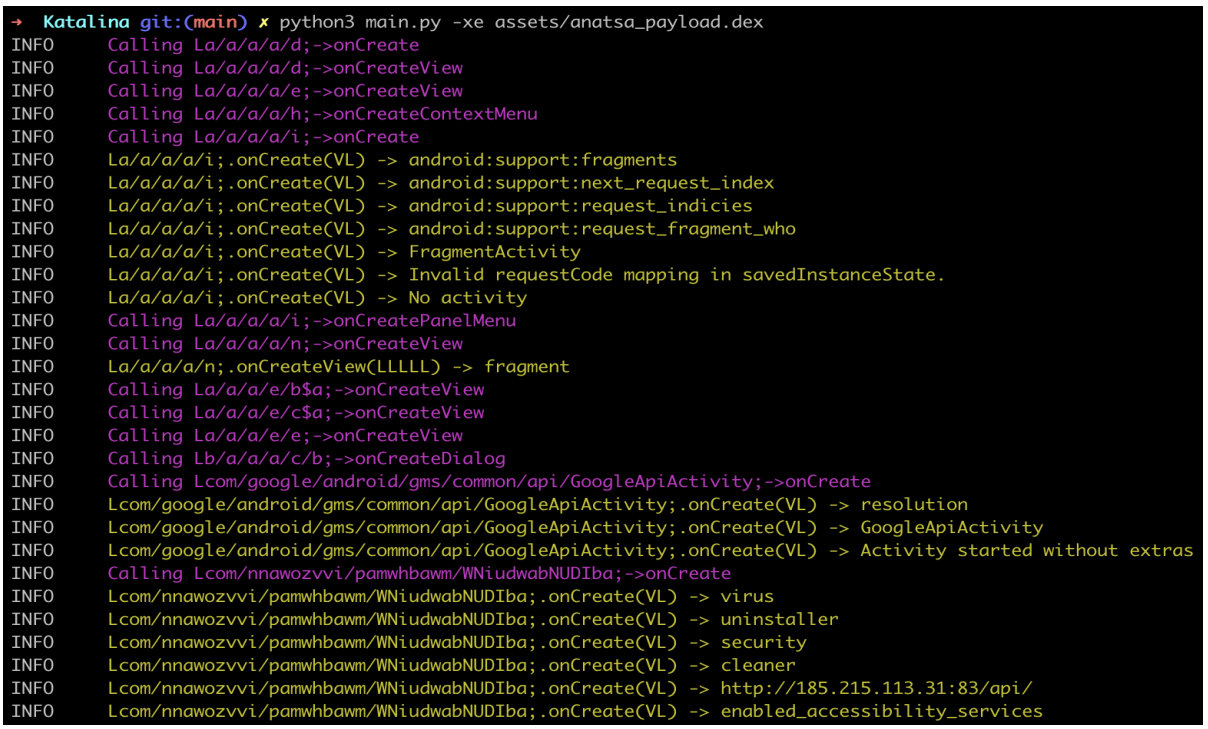


Fig 4.6 Katlina Output

4.2.5 Macneto

Concepts

Macneto contains two major modules: program (executable) analysis and machine learning, including PCA and deep learning. To demonstrate the capability of Macneto, we attempt to search for programs, mapping a program obfuscated by an obfuscator back to its original version. To deobfuscate programs, Macneto needs to learn the difference between the original binary and its obfuscated version. Thus, we need to program-analyze both versions of the binary. From step 1 to step 6, we will need to run them on both the original apk and obfuscated apk (or jars).

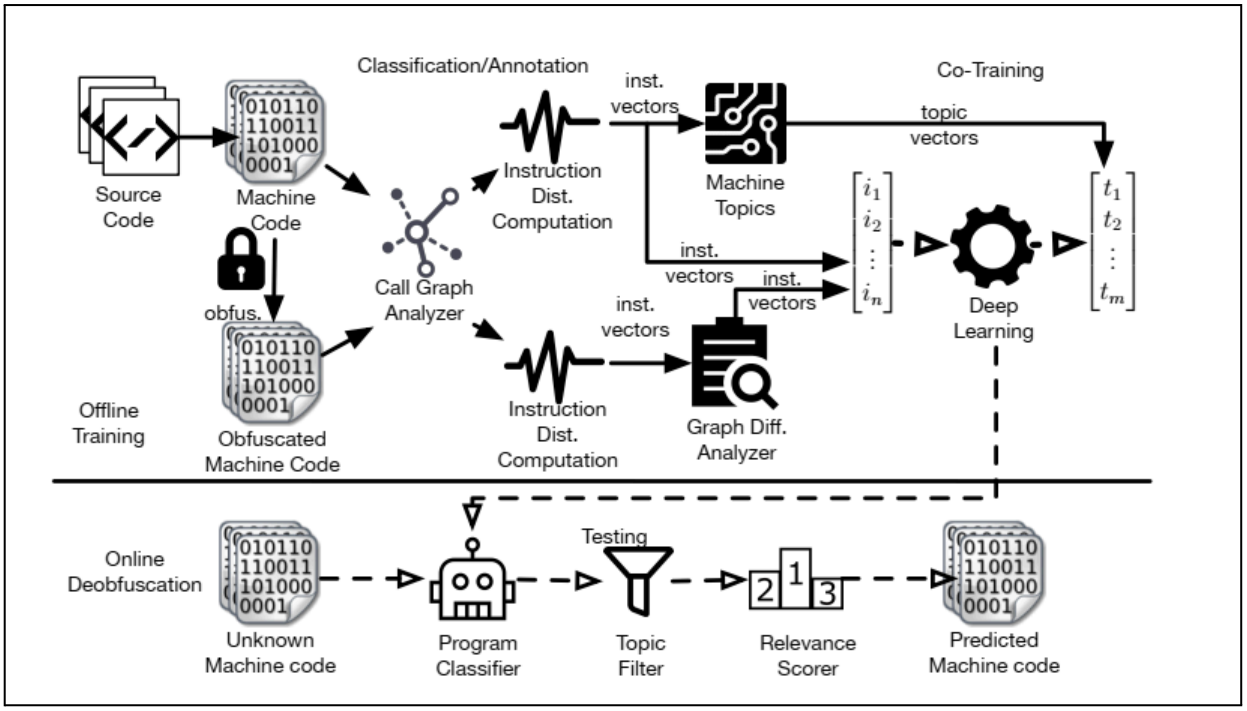


Fig. 4.7: The system architecture of MACNETO

The following steps, meticulously outlined by the tool's publisher, delineate the procedural framework for its execution

To facilitate the analysis, you can create a directory for analyzing your original binaries and create another directory for analyzing your obfuscated binaries. For learning the mapping between an executable and its obfuscated counterpart, we use sklearn, tensorflow and keras. For installing these frameworks

Steps

Step 0: Compilation

Our program analyzer is java based. For compiling our program analyzer, you will need to install Maven. Simply use the following command for compliation: mvn clean package

Step 1: Create database

```
java-cpmacneto-inst-analysis-0.0.1-SNAPSHOT.jar edu.columbia.cs.psl
.macneto.db.DocManager
```

This command will create a database db/macneto.db. You will have to create both databases for your original binaries and your obfuscated binaries. This means that you will have **n + 1** databases, where n is the number of your obfuscators.

Step 2: Generate call graphs

For computing the call graph for each apk, we leverage FlowDroid. The set-up of FlowDroid can be found on their github repo.

```
python macneto_container_v2/graph_gen.py -a YOUR_APKBASE -g callgraphs -e
macneto_container_v2/macneto-inst-analysis-0.0.1-SNAPSHOT.jar-f android/ platforms -t 16
-ext jar
```

For details of these options, please refer to graph_gen.py. Again, this command needs to run on both original apks and obfuscated apks/jars. -a can specify the directory of your apks. -g can specify where you want to store your callgraphs. We usually keep it as ./callgraphs. -t can specify the thread number. -ext can specify which types of binary you want to analyze.

Step 3: Compute instruction distribution

```
java -cp macneto-inst-analysis-0.0.1-SNAPSHOT.jar edu.columbia.cs.psl.
macneto.inst.MachineCodeDriver -c your/apk/location
```

This step computes the instruction distribution of each method in all of your apks. Again, this step needs to run on both original and obfuscated apks.

Step 4: Learning phase

```
python macneto_learn.py -d ORIGINAL/db/macneto.db -b OBFUSCATED/db/ macneto.db -t 32 -i 10000 -a deep
```

-t specifies how many principal components you want; -i specifies how many iterations you want; you can leave -a as it is. This step first compute PCA from the original executables, use it label the obfuscated counterparts and then learn the mapping through ANN.

Step 5: Search phase

```
python macneto_search.py -o ORIGINAL/db/macneto.db -b OBFUSCATED/db/ macneto.db -m 7
```

-m specifies the experiment (fold) number (m + 1). In our paper, we conduct k-fold analysis, where k = 8

OBSERVATIONS AND ANALYSIS

5.1 Android application forensics

1. **Ubiquity of Android Apps:**
    a. ***Observation***: The rapid proliferation of Android applications presents a significant challenge for forensic investigators due to the increasing volume of digital evidence.
    b. ***Analysis:*** The exponential growth of Android apps necessitates efficient tools and methodologies to handle the expanding workload effectively. This underscores the importance of scalable forensic analysis techniques.
2. **Impact of Obfuscation:**
    a. ***Observation:*** Obfuscation techniques, whether in benign or malicious apps, add complexity to forensic analysis by concealing code and structure.
    b. ***Analysis:*** The widespread use of obfuscation underscores its effectiveness in hindering analysis efforts. Detecting and deciphering obfuscated code requires advanced skills and tools, highlighting the need for continuous advancements in forensic techniques.
3. **Motives for Analysis:**
    a. ***Observation:*** Investigators analyze apps for evidence related to crimes or malicious activities, necessitating different approaches based on the motive.
    b. ***Analysis:*** Understanding the motive behind app analysis informs the level of scrutiny applied and the techniques employed. Forensic investigators must adapt their methodologies based on the suspected intent of the app.
4. **Detection and Deobfuscation:**
    a. ***Observation:*** Automated detection tools for obfuscation are limited, emphasizing the importance of manual deobfuscation techniques.
    b. ***Analysis:*** Manual deobfuscation is often necessary for a comprehensive analysis, requiring advanced skills in reverse engineering and code analysis. Automated

tools provide valuable insights but may not suffice for unraveling complex obfuscated code.

5. **Research Aim:**
    a. ***Observation:*** The paper aims to provide a comprehensive overview of obfuscation techniques and their forensic implications.
    b. ***Analysis:*** By synthesizing existing literature, the paper equips forensic investigators with knowledge and resources to navigate the complexities of obfuscated Android applications effectively. This holistic approach supports practitioners in addressing the challenges posed by obfuscation.
6. **Scope and Limitations:**
    a. ***Observation:*** The paper focuses on the Android platform, acknowledging the dynamic nature of obfuscation techniques.
    b. ***Analysis:*** While efforts have been made to provide a comprehensive overview, the evolving nature of the field and the Android platform's specificity may limit the generalizability of findings. Continuous research is necessary to keep pace with emerging techniques and challenges.
7. **Structure of the Paper:**
    a. ***Observation:*** The paper outlines a structured approach, facilitating a systematic exploration of the topic.
    b. ***Analysis:*** The organized framework allows readers to gain a comprehensive understanding of obfuscation techniques and their impact on forensic investigations. Each section contributes to a unified perspective, enhancing the paper's coherence and accessibility.

The impact of obfuscation on forensic investigations is profound, manifesting in two major implications. Firstly, obfuscation techniques serve to mislead automation, evading automatic detection and hindering forensic examination. Tactics such as evasion attacks can deceive machine-learning-based obfuscation detection methods, while simple techniques like repacking and manifest transformation invalidate application fingerprinting techniques, complicating the automated analysis further. Additionally, obfuscation techniques complicate manual analysis significantly, rendering some methods impractical, particularly in cases where the code is

encrypted or inundated with excessive junk code. Consequently, analysis becomes more challenging, necessitating advanced practitioners and exacerbating backlogs. To mitigate these challenges, various tools have been proposed to detect and sometimes deobfuscate applications. However, most automated approaches remain unreleased, contributing to the gap in effective obfuscation detection and deobfuscation, especially concerning native libraries. While recent advancements offer tools and techniques less impacted by obfuscation, the evolving complexity, including the possibility of layered obfuscation, underscores the ongoing challenges in malware analysis and deobfuscation. The conclusion consolidates these observations, presenting a condensed summary of the status quo of Android obfuscation, detection, and deobfuscation, advocating for a practitioner-centric approach to organizing tools for effective utilization. It highlights the increasing prevalence of obfuscation techniques in developer tools, emphasizing the need for advanced skills to counteract their effects, thereby emphasizing the critical role of dynamic analysis skills in future forensic investigations.

5.2 Deguard

The paper introduces a novel approach to deobfuscating Android applications using probabilistic models, leveraging a large corpus of existing Android programs, termed as "Big Code." By learning from this dataset, the approach develops a powerful probabilistic model capturing key features of non-obfuscated Android programs and utilizes it to suggest likely deobfuscations for new, obfuscated applications. Focusing on layout deobfuscation, particularly the renaming of program elements with semantically obscure identifiers, the approach addresses the challenges posed by layout obfuscation and presents a tool called DeGuard, which effectively reverses layout obfuscation performed by popular tools like ProGuard.

1. Effectiveness of DeGuard in Program Element Identification:

- The data analysis highlights the efficacy of DeGuard in reversing ProGuard's obfuscation techniques. A substantial number of initially obfuscated program elements are correctly predicted post-deobfuscation, indicating DeGuard's ability to reconstruct original names effectively.

- This underscores DeGuard's significance in enabling developers and analysts to comprehend the structure and functionality of obfuscated applications. It particularly aids in identifying third-party components, essential for both developers and security analysts.

2. Precision and Recall for Library Identification:

- DeGuard demonstrates notable precision and recall in predicting third-party libraries even after obfuscation by ProGuard. High precision suggests accurate identification of third-party libraries, while high recall implies successful identification of most existing libraries within the application.
- This precision and recall are crucial for developers understanding their codebases and analysts examining potentially malicious applications, enhancing both development and security efforts.

3. Utility in Security Analysis and Malware Deobfuscation:

- DeGuard's precision in predicting libraries and revealing sensitive data usage in malware samples underscores its significance in security analysis and malware detection.
- Security analysts can leverage DeGuard to uncover hidden functionalities, identify potential security vulnerabilities, and understand obfuscated application behavior, thereby aiding in malware analysis and reverse engineering tasks.

4. Limitations and Future Directions:

- While DeGuard demonstrates considerable effectiveness in deobfuscating program elements and revealing sensitive data usage, it may face challenges with more sophisticated obfuscation techniques employed by advanced malware.
- Future research directions could focus on enhancing DeGuard's capabilities to address additional obfuscation methods such as control flow obfuscation and string encryption.

Improvements in machine learning algorithms and heuristic approaches could further enhance DeGuard's effectiveness in malware analysis and reverse engineering tasks.

Limitations of DeGuard:

1. ***Sophisticated Obfuscation Techniques:*** While DeGuard demonstrates considerable effectiveness in deobfuscating program elements and revealing sensitive data usage, it may encounter challenges with more sophisticated obfuscation techniques employed by advanced malware. Such techniques could include control flow obfuscation and string encryption, which may pose difficulties for DeGuard in accurately reconstructing the original program structure.
2. ***Scalability and Learning:*** One of the limitations of DeGuard lies in its scalability in learning from large datasets. While it leverages a large corpus of existing Android programs (termed as "Big Code") for probabilistic modeling, scaling this approach to even larger datasets could pose computational challenges. Additionally, as the dataset grows, ensuring the quality and relevance of the learned features becomes increasingly complex.

Pros and Cons of DeGuard:

Pros:

1. ***Effective Reversal of ProGuard Obfuscation:*** DeGuard demonstrates effectiveness in reversing ProGuard's obfuscation techniques, accurately predicting a significant portion of obfuscated program elements. This enables developers and analysts to comprehend the structure and functionality of obfuscated applications, enhancing both development and security efforts.
2. ***Precision in Library Identification:*** DeGuard exhibits notable precision and recall in predicting third-party libraries, even after obfuscation by ProGuard. High precision ensures accurate identification of third-party libraries, while high recall implies successful identification of most existing libraries within the application. This precision is crucial for developers understanding their codebases and analysts examining potentially malicious applications.

3. ***Utility in Security Analysis:*** DeGuard's precision in predicting libraries and revealing sensitive data usage in malware samples underscores its significance in security analysis and malware detection. Security analysts can leverage DeGuard to uncover hidden functionalities, identify potential security vulnerabilities, and understand obfuscated application behavior, aiding in malware analysis and reverse engineering tasks.

Cons:

1. ***Challenges with Sophisticated Obfuscation:*** Despite its effectiveness, DeGuard may face challenges with more sophisticated obfuscation techniques employed by advanced malware. Techniques like control flow obfuscation and string encryption could hinder DeGuard's ability to accurately reconstruct the original program structure, limiting its effectiveness in certain scenarios.
2. ***Scalability Concerns:*** DeGuard's scalability in learning from large datasets could pose computational challenges, particularly as the dataset size grows. Ensuring the quality and relevance of learned features becomes increasingly complex with larger datasets, potentially impacting the accuracy and performance of DeGuard in practical applications.

In conclusion, while DeGuard offers significant advantages in reversing ProGuard obfuscation and aiding in security analysis, it also faces challenges related to sophisticated obfuscation techniques and scalability. Addressing these limitations could further enhance DeGuard's effectiveness in deobfuscation and malware analysis tasks.

5.3 Deoptfuscator

In the observation section, it's evident that highly obfuscated apps undergo significant structural changes compared to their original counterparts. The reduction in the number of methods indicates the impact of obfuscation, while the substantial increases in basic blocks and CFG edges suggest a more intricate control flow. This complexity is further reflected in the considerable rise in basic blocks and edges per method, highlighting the extensive restructuring induced by obfuscation techniques. Moreover, the increase in bytecode instructions signifies the

optimization process, which includes identifier renaming and method removal, ultimately leading to a 43% increase in the size of the dex file. Conversely, optimization with ReDex effectively reduces bytecode size to approximately 50% of the obfuscated one, demonstrating its efficacy in mitigating the effects of obfuscation. Similarly, moderately obfuscated apps exhibit similar trends but with smaller magnitudes, indicating a lesser degree of obfuscation. Despite this, optimization strategies still manage to reduce the size of the .dex file by approximately 76%, underscoring the effectiveness of optimization techniques in enhancing app efficiency and performance.

Limitations of Deoptfuscator

- The tool's effectiveness is primarily demonstrated in handling control-flow obfuscation by DexGuard, limiting its applicability to other obfuscation techniques.
- Arbitrary removal of opaque variables poses a risk of errors, especially in the presence of anti-tampering mechanisms, necessitating cautious implementation.
- The study's limited experimentation with a narrow range of apps and smaller sample sizes may impact the generalizability of results, highlighting the need for broader validation.

Pros and Cons of Deoptfuscator

- **Pros:**
    - The tool adeptly handles control-flow obfuscation by DexGuard, evidenced by its ability to reduce complexity and restore code structure.
    - Method-by-method similarity analysis provides detailed insights into deobfuscation effectiveness, facilitating targeted improvements.
    - Optimization with ReDex results in substantial reductions in obfuscation and bytecode size, enhancing app performance and efficiency.
    - Flexibility in deobfuscation aggressiveness allows users to tailor the process to their specific needs, balancing between effectiveness and resource utilisation.
- **Cons:**

- Limited scope to handle obfuscation techniques from other tools like DashO or Allatori, potentially restricting its applicability in diverse scenarios.
- Effectiveness may vary depending on the complexity and diversity of obfuscation techniques employed, necessitating further validation with a broader range of apps.
- Uncertainty regarding the execution of deobfuscated apps with anti-tampering protection raises concerns about practical applicability in real-world scenarios.

Combined Results and Implications:

- While the tool exhibits proficiency in addressing control-flow obfuscation by DexGuard, its applicability may be constrained by the specificity of obfuscation techniques.
- Optimization with ReDex and Deoptfuscator shows promising results in reducing obfuscation and increasing similarity to the original app, underscoring potential avenues for further research and development in Android app deobfuscation.
- Addressing limitations related to obfuscation diversity, error handling, validation scope, and anti-tampering concerns is crucial for enhancing the tool's effectiveness and practical utility in real-world scenarios.

5.4 Linear Mixed Boolean-Arithmetic Expressions

The SiMBA algorithm, designed as a Python-based tool for computing linear Multiple Binary Algebra (MBA) expressions, introduces a novel approach to directly evaluate MBAs for all possible truth value combinations without transforming inputs into 1-bit space. By computing signature vectors from all possible combinations of variable values, SiMBA eliminates the need for decomposing the MBA into bitwise expressions and coefficients, potentially enhancing performance, especially for complex MBAs with numerous variables. It derives linear combinations of base bitwise expressions for 't' variables using a generic approach, simplifying

the process without extensive linear algebra. Employing various simplification strategies based on signature vector uniqueness, SiMBA aims to find simpler expressions, as validated through experiments demonstrating its correctness, universality, and competitive performance compared to existing methods like MBA-Blast and MBA-Solver, particularly for MBAs with higher variable counts. Additionally, SiMBA proves capable of handling encoded MBAs using affine functions, showcasing its versatility and robustness. In conclusion, SiMBA offers a novel and efficient approach to linear MBA computation, combining simplicity, efficiency, and adaptability, making it a promising tool for various MBA simplification and analysis applications.

Limitations:

1. ***Complexity for Higher Variables:*** While SiMBA performs well for expressions with a moderate number of variables, it may struggle with complexity for higher numbers of variables. This complexity can affect both runtime and the ability to find simpler solutions.
2. ***Incomplete Simplification:*** In some cases, SiMBA may not find the simplest expression due to limitations in its refinement attempts. This can result in expressions that are more complex than necessary.
3. ***Verification Challenges:*** Verifying the equivalence of simplified expressions can be computationally intensive, especially for more complex expressions. This may limit the scalability of the algorithm for large datasets.

Pros:

1. ***Immediate Evaluation:*** SiMBA's approach of immediately evaluating the entire linear MBA allows for faster processing compared to methods that require decomposing the expression into terms.

2. ***Generic Approach:*** SiMBA uses a generic approach to transform input expressions into linear combinations of base expressions, making it applicable to a wide range of scenarios without the need for specific input structures.
3. ***Competitive Runtimes:*** SiMBA demonstrates competitive runtimes compared to other simplification tools like MBA-Blast and MBA-Solver, especially for expressions with a moderate number of variables.

Cons:

1. ***Limited Scalability:*** SiMBA may face scalability challenges for highly complex expressions or datasets with a large number of MBAs. The algorithm's performance may degrade significantly under these conditions.
2. ***Complexity Handling:*** The algorithm's handling of complexity, especially for expressions with a high number of variables, may lead to suboptimal solutions or longer runtimes.
3. ***Verification Overhead:*** Verifying the correctness of simplified expressions can introduce additional computational overhead, particularly for expressions that require extensive verification steps. This overhead may impact overall performance and efficiency.

Combined Result:

SiMBA offers a promising approach to simplifying linear MBA expressions, leveraging immediate evaluation and a generic transformation method. While it demonstrates competitive runtimes and versatility, it may encounter challenges with scalability and handling complexity for highly complex expressions. Additionally, verification overhead and the potential for incomplete simplification are important considerations. Overall, SiMBA presents a valuable tool for simplifying linear MBAs but may require further refinement to address scalability and complexity concerns effectively.

5.5 Katalina

Observation and Analysis:

Katalina, an open-source Android string deobfuscator, addresses the challenge of analyzing Android malware by providing a free tool for researchers to deobfuscate strings found in Android malware samples. Unlike commercial solutions, Katalina democratizes malware analysis by offering a novel approach to deobfuscation without the financial barrier. It utilizes a Python-based environment that emulates Android bytecode execution, enabling users to efficiently deobfuscate strings hidden behind complex logic, thereby accelerating the reverse engineering process. By executing bytecode instructions in a controlled environment, Katalina aims to streamline the analysis of Android malware, particularly those employing advanced obfuscation techniques. The tool's ability to automate the deobfuscation process not only reduces the manual effort required but also enhances the scalability of malware analysis efforts. Katalina leverages the legacy of Dalvik bytecode execution and integrates mock framework calls to simulate Android runtime behavior, providing a high-level overview of Android bytecode execution. The tool's simplicity and effectiveness lie in its ability to parse .dex files using KaitaiStruct and execute bytecode instructions to retrieve deobfuscated strings, thereby aiding in malware fingerprinting and IoC generation. Katalina's modular design allows users to execute specific functions, define denylists to skip irrelevant code, and potentially support multidex files in the future, enhancing its usability and versatility.

Limitations:

1. Lack of support for multidex files limits its applicability to larger Android applications distributed across multiple dex files.
2. Performance overhead may be introduced, especially with complex malware, due to the reliance on emulation for bytecode execution.
3. Evolving obfuscation techniques in Android malware may pose challenges to Katalina's effectiveness over time.

Pros:

1. Provides a free and open-source alternative to commercial deobfuscation tools, making malware analysis more accessible to the security community.

2. Offers modular design and flexibility, allowing users to customize the deobfuscation process according to specific requirements.
3. Enables efficient analysis of Android malware by emulating most instructions, same-class method invocations, static fields/method invocations, and string APIs.

Cons:
1. Limited support for certain functionalities such as multidex files, iterator APIs, arrays APIs, and cross-class non-static method invocations and fields.
2. Potential issues with I/O operations and Windows support, which may require users to use Windows Subsystem for Linux (WSL) for proper functionality.

In summary, Katalina presents a promising solution for Android malware analysis with its free and open-source platform. While it offers innovative string deobfuscation techniques and emulates various Android bytecode instructions effectively, it faces limitations such as lack of multidex support and potential issues with I/O operations and Windows compatibility. Despite these drawbacks, Katalina's modular design and focus on string obfuscation make it a valuable tool for security researchers, providing insights into hidden malicious behaviors and enhancing overall threat detection capabilities.

5.6 Macneto

Observation and Analysis:
The paper evaluates the performance of MACNETO, a system designed for deobfuscating programs, using two main metrics: precision and Top@K. Precision measures how accurately MACNETO can recover the original version of obfuscated methods, while Top@K measures the performance of MACNETO in identifying the original method within the top K results.

1. **Evaluation Metrics:**
    - Precision (Equation 10): MACNETO achieves a precision of 96.29% for Top@1, 99.31% for Top@3, and 99.86% for Top@10 when evaluated against ProGuard-obfuscated programs. This indicates MACNETO's high accuracy in deobfuscating methods transformed by a lexical obfuscator.

- Top@K (Equation 11): MACNETO's performance is further evaluated using Top@K, where K is set to 1, 3, and 10. The results show consistent high performance across different values of K.

2. **Comparison with DeGuard:**
    - MACNETO outperforms DeGuard, a previous tool, in terms of precision. While MACNETO achieves a precision of 96.29% for Top@1, DeGuard's precision on deobfuscating method names alone is reported to be 66.59%. This highlights MACNETO's superiority in accurately recovering method names from obfuscated code.
3. **Performance Against Different Obfuscation Techniques:**
    - MACNETO's performance is evaluated against Allatori-obfuscated code, which involves control flow changes and string encryption/decryption. The results demonstrate MACNETO's ability to achieve up to 91% precision in deobfuscating methods obfuscated using control and data transformation techniques.
4. **Effectiveness of Program Classifier and Graph Diff Module:**
    - MACNETO's program classifier filters out methods that do not share similar classifications with obfuscated methods, enhancing its deobfuscation performance.
    - The graph diff module enables MACNETO to identify injected decryption methods accurately, leading to improved deobfuscation results.
5. **Limitations:**
    - MACNETO relies on the callgraph of the app, limiting its ability to deobfuscate methods not included in the callgraph.
    - The graph diff module may not handle obfuscation techniques involving randomly generated methods effectively.

Pros and Cons:

Pros:

- High precision rates: MACNETO achieves high precision rates, especially when compared to existing tools like DeGuard.
- Versatility: MACNETO demonstrates effectiveness against various obfuscation techniques, including lexical obfuscation and control/data transformation.
- Utilization of advanced techniques: MACNETO leverages deep learning and graph diff algorithms to enhance its deobfuscation capabilities.

Cons:

- Dependency on callgraph: MACNETO's reliance on the callgraph may limit its effectiveness in scenarios where methods are not included in the callgraph.
- Limitations in handling certain obfuscation techniques: MACNETO may face challenges in handling obfuscation techniques involving randomly generated methods or complex control flow transformations.

Overall, MACNETO shows promising results in deobfuscating obfuscated programs, offering high precision and versatility in handling different obfuscation techniques. However, its effectiveness may be limited by dependencies on specific program structures and challenges in handling certain types of obfuscation.

CONCLUSION

In conclusion, the examination of various deobfuscation techniques, including DeGuard, Deoptfuscator, SiMBA, Katalina, and MACNETO, presents a nuanced understanding of their strengths and limitations in deciphering obfuscated code within drone Android applications.

DeGuard exhibits commendable proficiency in reversing ProGuard obfuscation, particularly in predicting third-party libraries with high precision and recall rates, thus facilitating improved comprehension of application structures and functionalities. However, challenges persist with sophisticated obfuscation techniques and scalability concerns.

Deoptfuscator demonstrates effectiveness in handling control-flow obfuscation, contributing to code structure restoration and optimizing app performance through ReDex. Nevertheless, its limited applicability to other obfuscation techniques and the potential risk of errors require careful implementation consideration.

SiMBA introduces an efficient approach to linear MBA computation, boasting competitive runtimes and adaptability. Despite its efficiency, challenges arise with handling complexity and verifying expression equivalence, impacting its scalability for larger datasets.

Katalina serves as a valuable free and open-source alternative for malware analysis, efficiently deobfuscating strings and accelerating reverse engineering processes. However, its lack of multidex support and potential performance overheads warrant attention, especially with complex malware.

MACNETO showcases high precision rates in deobfuscating obfuscated programs, exhibiting versatility against various obfuscation techniques. Yet, dependencies on callgraph and challenges with certain obfuscation techniques necessitate further refinement for broader effectiveness.

In conclusion, it's evident that no single deobfuscation method stands as universally superior, with each technique showcasing specific strengths and weaknesses tailored to different types of obfuscation. Effective deobfuscation often demands the integration of multiple methods to comprehensively address various obfuscation strategies. Furthermore, not every application opts for obfuscation; while some prioritize transparency and ease of debugging, others employ heavy

obfuscation to safeguard proprietary algorithms or sensitive data, depending on developers' objectives and security concerns. Crucially, deobfuscators play a critical role in detecting malware within drone applications by reversing obfuscation techniques and revealing the original code. This allows analysts to identify and analyze malicious components effectively, ensuring the safety and security of drone operations by uncovering suspicious code patterns and behaviors.

REFERENCES

[1] Benjamin Reichenwallner & Peter Meerwald-Stadler Denuvo GmbH, “Efficient Deobfuscation of Linear Mixed Boolean-Arithmetic Expressions”.[Online]. Available: https://www.researchgate.net/publication/363564157_Efficient_Deobfuscation_of_Linear_Mixed_Boolean-Arithmetic_Expressions

[2] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, “Code Relatives: Detecting Similarly Behaving Software,” in 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2016. New York, NY, USA: ACM, November 2016, pp. 702–714, artifact accepted as platinum. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950321

[3] Fang-Hsiang Su, Jonathan Bell George Mason , Gail Kaiser , Baishakhi Ray , Deobfuscating Android Applications through Deep Learning”.[Online]. Available: https://mice.cs.columbia.edu/getTechreport.php?techreportID=1632

[4] Xiaolu Zhang a , Frank Breitinger b, Engelbert Luechinger c, Stephen O'Shaughnessy, Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations ” in Forensic Science International: Digital Investigation 39 (2021) 301285. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281721002031

[5] Benjamin Reichenwallner & Peter Meerwald-Stadler Denuvo GmbH, “Efficient Deobfuscation of Linear Mixed Boolean-Arithmetic Expressions”.[Online]. Available: https://www.researchgate.net/publication/363564157_Efficient_Deobfuscation_of_Linear_Mixed_Boolean-Arithmetic_Expressions

[6] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, “Statistical Deobfuscation of Android Applications,” in 23rd ACM Conference on Computer and Communications Security, ser. CCS 2016. New York, NY, USA: ACM, October 2016, pp. 343–355. Online]. Available: http://doi.acm.org/10.1145/2976749.2978422

[7] Proguard. [Online]. Available: https://www.guardsquare.com/en/proguard

[8] C. S. Collberg and C. Thomborson, “Watermarking, tamper-proffing, and obfuscation: Tools for software protection,” IEEE Trans. Softw. Eng., vol. 28, no. 8, pp. 735–746, Aug. 2002.