Compiler Provenance Attribution using Machine Learning

1st Yousef Mohamed
Ottawa ID: 300389908
Group 17

2nd Hossam Elqersh
Ottawa ID: 300389399
Group 17

3rd Ahmed Abdelaal
Ottawa ID: 300389360
Group 17

4th Mohamed Abdelaal
Ottawa ID: 300389359
Group 17

***Abstract*—The identification of compilers used to create distinct binary structures is a crucial task, offering valuable insights into security vulnerabilities and intellectual property concerns. However, the complexity of these binaries demands advanced tools for accurate attribution. Leveraging machine learning, we propose the development of models capable of distinguishing patterns in binaries to determine their compiler origins. After experimenting with multiple machine learning models to identify the most accurate one, we achieved 100% accuracy and F1 Score for compiler detection and 71% F1 Score for predicting the optimization level. To generate data, we will utilize executables from a designated dataset, extracting features and strings using a disassembly tool. The gathered data will undergo preprocessing and feature engineering before being used for training and testing the models.**

I. Introduction

In the realm of digital forensics and binary analysis, a comprehensive understanding of the origins and characteristics of program binaries is paramount. These intricate and diverse binary structures stem from the distinct optimization techniques and settings applied by different compilers. It is crucial to acknowledge that while optimization is pivotal for enhancing performance and efficiency, it may introduce unintended consequences. Higher optimization levels, aimed at streamlining code execution, can inadvertently introduce bugs and vulnerabilities. The optimization process has the potential to obscure the original intent of the code, complicating efforts to identify and address potential security threats.

Additionally, compilers exhibit unique traits specific to each, offering valuable insights ranging from pinpointing potential security issues to addressing concerns related to intellectual property. Forensic investigations often revolve around program binaries, encompassing various issues such as copyright violations and malware analysis.

Moreover, it is noteworthy that malware authors frequently employ specific compilers with predefined optimization levels. Recognizing these compiler attributes holds a pivotal role in the malware analysis process, significantly contributing to the understanding and mitigation of malicious software. However, analyzing program binaries is far from straightforward. It's complicated by the absence of the high-level information typically found in source code and made more challenging by the many compiler variations in use, including different versions and optimization levels. Compiler provenance, a key aspect of this analysis, includes information about the compiler family, version, optimization level, and compiler-related functions.

The previous work in machine code analysis, each with its strengths and limitations, introduces tools such as o-glassesX, which exhibits proficiency in recognizing software or compiler families and common machine language instructions. However, it encounters difficulties when attempting to understand more comprehensive details about computer programs. BinProv excels in compiler identification and optimization level classification but is constrained by assumptions about de-obfuscated code and a single CPU architecture focus. DIComP and BinComp, while innovative in compiler provenance extraction, grapple with challenges like reproducing vulnerabilities and reliance on unstripped code.

Our proposed method involves leveraging advanced tools and methods, specifically employing machine learning to construct models capable of discerning patterns within program binaries, thereby unveiling their compiler origins. Instead of relying on a singular machine learning model, we intend to explore multiple options to identify the most accurate one. To collect the necessary data for this project, we will utilize executable files from the recommended dataset 'BinComp,' authored by PariaSh in 2015 and accessible on GitHub [1]. Employing a disassembly tool, we will extract essential features and strings from these binaries. Subsequently, we will prepare the gathered data for training and testing the machine learning models, including Logistic Regression, SVM, Random Forest, Decision Tree, Gradient Boost, and NN. This involves preprocessing, feature engineering, and NLP transformations like TF-IDF, along with incorporating features extracted from strings. This comprehensive approach aims to tackle the challenges associated with determining compiler provenance, thereby advancing the fields of digital forensics and program binary analysis.

II. State-of-the-art review

In this section, we present a comprehensive review of the state of the art in Compiler Provenance Attribution using Machine Learning. This review encompasses significant findings and contributions from relevant research papers in the field.

A. "o-glassesX: Compiler Provenance Recovery with Attention Mechanism from a Short Code Fragment"

In this research [2] a novel method for comprehending computer program code written in machine language is introduced. Imagine machine language as the secret code of software, often making it challenging to determine which software created it or its purpose. The new method employs advanced learning techniques to recognize and understand this code. The researchers tested the method, and it consistently performed well, accurately identifying the software or compiler family responsible for the code. They also discovered its ability to identify common machine language instructions used in various software, like recognizing frequently used words by an author. This intelligent method has the potential to greatly aid in understanding and safeguarding against malicious software or deciphering code with limited information.

Nevertheless, some challenges remain, particularly in comprehending broader information about computer programs and their authors.

As previously discussed, the author's goal is to develop a method for analyzing binary machine code, determining its source compiler family, and optimizing levels. The author seeks to address the common challenge of extracting this information, especially in the context of malware and other executables analysis. The solution presented in the paper introduces a novel approach that utilizes an attention mechanism to classify sequences of code instructions. This approach not only achieves high accuracy in classification but also offers insights into the contribution of individual instructions to the classification outcome. Additionally, it enables the automatic extraction of typical instructions for each compiler class, shedding light on the unique attributes of different compilers.

While the proposed method exhibits significant strengths in recognizing compiler origin and optimization levels, it does come with some limitations. For instance, it assumes that machine code is already de-obfuscated, which may not always hold true in real-world scenarios. Moreover, the method is designed for one CPU architecture at a time, necessitating prior identification of the architecture when dealing with mixed inputs. Furthermore, its direct applicability to broader goals of program origin and author identification from binary code may be limited. Despite these constraints, the approach presents a promising solution for specific binary analysis tasks.

The strength of the author's work lies in its effective recognition of the source compiler family and optimization levels from binary machine code. This method achieves an impressive accuracy rate, making it a valuable tool for analyzing malware and other executables. Additionally, the approach not only classifies code sequences but also reveals the contribution of individual instructions to the identification, providing a deeper understanding of how compilers generate code. Furthermore, it automatically extracts typical instructions for each compiler class, assisting analysts in discovering unique compiler characteristics. However, some weaknesses in this approach should be noted. One notable limitation is its reliance on the assumption that the machine code is already de-obfuscated, which might not hold true in many real-world scenarios, posing a challenge for its direct application to obfuscated code. Another drawback is its restriction to a single CPU architecture at a time, requiring prior identification of the architecture when dealing with mixed inputs. Furthermore, the method may not directly address broader goals of program origin and author identification from binary code. Nonetheless, the approach offers a robust solution for specific binary analysis tasks, particularly in recognizing compiler origins and optimization levels.

B. "BinProv: Binary Code Provenance Identification without Disassembly"

The paper [3] introduces an end-to-end deep-learning framework for identifying compilation provenance from binary code. It employs a BERT-based embedding model to capture contextual semantics, eliminating the need for disassemblers and manual feature extraction. BinProv is architecture-agnostic and can distinguish between four optimization levels. It addresses three main challenges: the absence of assembly code, the difficulty of capturing key information from sparse binary semantics, and the complexity of the BERT model by using a transfer learning strategy. BinProv can be fine-tuned for various tasks, including compiler identification and optimization level identification.

They pre-train the model to understand contextual semantics in byte sequences using a Masked Language Modeling task, and then they fine-tune it for specific provenance tasks like compiler and optimization level identification. BinProv can also be adapted for tasks such as architecture and function type identification. To address the challenge of accurately identifying provenance at different levels, they use a majority voting mechanism based on real-world project observations, where most projects tend to use the same optimization level for their binary files. This approach improves identification accuracy for both individual functions and entire binary files.

In this analysis, the authors compared binary length and the similarity of byte sequences from 235 binaries in the BINKIT dataset using the Normalized Compression Distance (NCD). The results revealed that the number and types of optimization flags used at different optimization levels have a significant impact on the differences between byte sequences. Notably, the difference between optimization levels O0 and O1 is the most pronounced, with an NCD greater than 0.92, and the binaries under O1 are the shortest. This difference is primarily due to the presence of 43 optimization flags in O1, which effectively reduce the code size. In contrast, byte sequences are more similar between O2 and O3, with an NCD of less than 0.58, and the binary length under O2 and O3 becomes longer. This is because O3 adds only 15 flags but introduces more complex constructs, such as overlapping blocks/instructions, inline data, and tail calls, leading to an increase in code size.

The architecture of BinProv comprises four key stages: input pre-processing, embedding generation, classification, and joint inference. During pre-processing, BinProv captures and

restructures the raw byte sequences from a given binary file. Subsequently, BinProv converts these inputs into embedding vectors using a BERT-based embedding network, which refines the semantic understanding of the byte sequences. With the generated embedding representations, BinProv employs a classification network to determine the provenance of these sequences. The process follows a pretrain-finetune strategy. By initially identifying provenance at the sequence level, BinProv combines multiple sequences to ascertain provenance at the binary-level and function-level.

In compiler identification, all methods achieve over 95% accuracy, with BinProv, even without fine-tuning, maintaining a small 2% gap with other methods. This accuracy is due to distinct compiler patterns, like "push ebp" for GCC and specific Clang patterns. In optimization level classification, all methods perform well, with BinProv without fine-tuning (98.9%) outperforming others, and "Origin" being the least accurate (96.14%). The key factor here is the significant differences between high and low optimization levels, with various optimization flags creating substantial changes in binary constructs, such as size variations.

In fine-grained optimization level classification, BinProv excels compared to the baseline methods, Origin and O-glassesX, which struggle with four-class classification (O0/O1/O2/O3). While Origin and O-glassesX experience a significant drop in accuracy when distinguishing between O0/O1 and O2/O3, BinProv maintains high accuracy, achieving 91% for O0/O1/O2/O3 and 98.5% for O0/O1. Even in the challenging task of distinguishing O2 from O3, BinProv outperforms other methods with an accuracy of 83.64%, surpassing them by 20%. The most demanding aspect of this classification is distinguishing O2 from O3 due to subtle differences in optimization flags. Precision, recall, and F1 scores vary for different optimization levels, with O2 and O3 showing lower values because of their similarity. Overall, BinProv demonstrates significant advantages. Firstly, it doesn't rely on auxiliary information, avoiding errors from disassemblers and feature engineering complexities. While disassembler accuracy drops with higher optimization levels, BinProv maintains better accuracy, even with obfuscated binaries. Secondly, BinProv offers finer-grained classification of optimization levels, distinguishing between the most commonly used O2/O3 with superior accuracy compared to baselines. However, BinProv has limitations, such as potential issues with x86 sequences, non-code bytes mingled in code sections, and susceptibility to new obfuscation techniques. Future work may involve identifying additional optimization flags and extending BinProv's applications, such as detecting third-party components in binary packages through provenance analysis. Finally, This paper introduces BinProv, a comprehensive framework for identifying compilation provenance in binary code. BinProv employs a BERT-based embedding model to capture contextual semantics in binary data. It relies solely on byte sequences as input, eliminating the need for disassemblers. BinProv achieves high accuracy in compiler and optimization level identification, particularly excelling in identifying high optimization levels (O2/O3). It employs majority voting to determine provenance at the function and binary levels, surpassing existing methods with 96.85% and 99.8% accuracy, respectively. The framework utilizes pretraining and fine-tuning through transfer learning with a BERT-based model to provide a fundamental understanding of binary semantics, making it adaptable to various tasks, including compiler helper function identification and enhancing binary code similarity detection.

C. "DIComP: Lightweight Data-Driven Inference of Binary Compiler Provenance with High Accuracy"

This research [4] is in the domain of binary analysis, with a primary focus on the implications of code compilation for software security and vulnerability testing. The authors utilized a comprehensive dataset derived from several open-source software projects, amassing over 28,000 binaries. These binaries were compiled using various optimization levels (-O0 to -O3 and -Os) and versions of two of the most widely used compilers, GCC (versions 5 to 9) and Clang (versions 3.9 to 9). The tool's strengths are in its high accuracy rates. For instance, DIComP achieved a 100% accuracy rate in distinguishing between GCC and Clang compilers based on the distribution of mnemonics and registers. However, challenges arise when differentiating between optimization levels like -O2, -O3, and -Os, especially for the Clang compiler, where the appearance of binaries is more similar. The research also highlighted the use of tools like BinDiff for binary similarity measurement, pointing out the impact of optimization levels on binary similarity. Furthermore, the authors acknowledged the challenges in reproducing vulnerabilities due to the absence of essential building configurations and the implications of overlooking the compiler's impact, which can lead to skewed type inference outcomes. While DIComP uses the Multi-Layer Perceptron (MLP) for its lightweight nature, the authors discuss the limitations of other models like RNN and CNN, which demand substantial computational resources. In their methodology, the authors used objdump for binary disassembly, achieving 99.4% accuracy. The assembly code was then split into mnemonics and operands for further analysis. The research also mentioned the challenges faced when distinguishing between different versions of compilers, especially Clang, where the similarity between versions was notably high. When compared to related works, DIComP showcased its superiority in several aspects. For instance, while previous works like i2v RNN, HIMALIA, and BinEye leveraged various methods, none could meet the balance between accuracy and efficiency as effectively as DIComP. The paper also highlighted the limitations of relying solely on control flow for distinguishing binaries compiled from different optimization levels, emphasizing the need for a more comprehensive feature set.

D. "BinComp: A stratified approach to compiler provenance Attribution"

This paper [5] introduces BinComp, a novel approach that aims to address the limitations of existing methods in

compiler provenance extraction. It is a crucial field in software analysis, impacting tasks like authorship attribution and clone detection. The paper identifies three key issues with previous approaches: the reliance on generic feature templates, the demand for substantial datasets and computational resources, and the vulnerability of exact matching algorithms to slight code differences.

BinComp's innovative approach can be summarized in two main motivations. First, it diversifies feature analysis by considering a combination of syntactic, semantic, and structural attributes, avoiding the trap of irrelevant code details. Second, it explicitly labels compiler-related functions, improving authorship attribution, function recognition, and clone detection by focusing on pertinent code sections, thus reducing errors.

The stratified architecture of BinComp comprises three layers. The first layer analyzes the code's structure, creating a "Compiler Transformation Profile" (CTP) and "compiler tags" (CT) that serve as distinctive fingerprints for recognizing compiler-induced changes. The second layer focuses on compiler-related functions and generates feature lists that describe these functions. The third layer explores the meaning and relationships within the code, producing the "Compiler Constructor Terminator" (CCT) graph and the "Annotated Control Flow Graph" (ACFG), which assist in determining not just the compiler used, but also the specific version and optimization techniques applied.

The novelty of BinComp lies in its ability to combine prior knowledge of compiler code transformations with incremental learning. This combination allows it to adapt to evolving compiler versions and changing code, setting it apart from traditional approaches.

BinComp's strengths become evident in its outstanding performance during evaluation. It achieved an impressive average accuracy of 0.97 in detecting the Visual Studio compiler, surpassing the ECP approach's accuracy of 0.93. Its effectiveness in handling variations in compiler versions and optimization levels is commendable, although pinpointing specific compiler versions, as demonstrated with the XCode compiler, remains a challenge.

While BinComp excels in many aspects, certain weaknesses are worth noting. It assumes that the code is already deobfuscated, limiting its application to obfuscated code. Additionally, it relies on the code being unstripped, which may not always be the case in practice. The evaluation was predominantly conducted on the Intel x86/x86-64 architecture, raising questions about its applicability to other architectures. Moreover, the focus on compiled C++ programs during the evaluation prompts discussions about its effectiveness in handling other programming languages and software types.

In summary, BinComp emerges as a powerful and innovative approach in compiler provenance extraction, offering a diverse feature set, fine-grained analysis, and adaptability to changing compiler landscapes. While it has some limitations related to code obfuscation, stripping, architecture, and specific programming languages, these challenges present opportunities for future research and refinement.

E. "Revisiting Lightweight Compiler Provenance Recovery on ARM Binaries"

In their paper [6] the authors present a comprehensive approach to predict compiler provenance for stripped ARM binaries. This approach builds upon and extends the techniques used in DIComP [4], a shallow learning model designed for compiler provenance recovery in x86-64 binaries. The primary goal is to efficiently and accurately recover compiler configuration properties for ARM binaries, and this is achieved through a set of key steps and adaptations:

1. Preprocessing and Disassembly: Similar to DIComP, the authors use the objdump utility to disassemble all executable sections of the input binary. This results in the binary being converted into its corresponding assembly code.

2. Profiling Register Usage: The authors collect relative frequency distributions of source and destination operands over all general-purpose registers. Notably, they adapt this process for 64-bit ARM assembly, where registers can be referenced by either all 64 bits or their lower 32 bits. This distinction is made because compiler usage may differ based on the bit size, e.g., for holding pointers or integers. They also incorporate the ratio of references between floating-point (fp) and stack pointer (sp) usage as an additional feature.

3. Profiling Opcodes with TF-IDF: In contrast to DIComP, which manually selects a fixed set of opcodes for each classification task, the authors adopt a more automated approach using TF-IDF (term frequency-inverse document frequency). They treat the opcode sequences in binaries as documents, where opcodes are words. This method quantifies the relative importance of each opcode, identifying those that stand out as distinctive features for compiler provenance.

4. Classification with Linear SVMs: The authors combine the frequency distributions of register usage and the TF-IDF scores for each binary, creating a feature vector. They then employ a hierarchy of four linear-kernel Support Vector Machines (SVMs) with L1 penalty to perform classification. The SVMs predict the compiler family, version, and optimization level, following a structure similar to DIComP's classifier.

5. 10-Fold Cross-Validation: The authors employ a 10-fold cross-validation technique and set class weights inversely proportional to class frequency to combat overfitting. In each fold, they first classify the compiler family and optimization level in parallel. This family prediction then guides whether the binary should be further classified for a specific compiler version (e.g., GCC or Clang).

The authors' approach results in a tuple of predicted compiler family, version, and optimization labels for each binary, and this model is applied to experimental ARM binary corpora to measure its accuracy and efficiency. The authors emphasize that their solution not only achieves accuracy on par with deep-learning baseline models but also offers significantly improved training and evaluation speeds, which are crucial for the scalability of their approach to large repositories of ARM binaries.

This approach, building upon DIComP's foundational model, addresses the challenge of recovering compiler prove-

nance in ARM binaries using lightweight, interpretable methods and is a valuable contribution to the field of binary analysis.

The authors aimed to recover compiler provenance from unknown and stripped ARM binaries using a model based on DIComP. Their model achieved impressive accuracy results, with an accuracy of approximately 0.9988 for the compiler family and around 0.9167 for the optimization level on 64-bit ARM binaries. Overfitting concerns observed in CNN-based models were addressed by employing a lightweight approach, which led to a significant increase in speed while maintaining similar accuracy. Their model was also able to distinguish between old and new compiler versions with an accuracy of 0.9664 for gcc and 0.8522 for Clang. When applied to 32-bit ARM binaries, the model maintained its high accuracy, achieving an accuracy of 0.9920 for the compiler family and 0.7860 for the optimization level with five labels. Further simplification of the optimization level labels resulted in an accuracy of 0.8560. When tasked with detecting the presence or absence of optimization, the model's accuracy reached 0.9929. Additionally, when distinguishing CompCert binaries from those compiled with other compilers, the model achieved a high accuracy of 0.9934. Furthermore, the feature weights provided valuable insights into the differences in code generation between CompCert and other compilers. When applied to a large corpus of ARM binaries from various Linux distributions, the model showed a strong prediction affinity towards gcc, while aligning with the specific optimization preferences of the Linux distributions. This work demonstrates the model's effectiveness in recovering compiler provenance for ARM binaries, offering speed, interpretability, and adaptability across diverse scenarios.

The strength of this study lies in its successful development of an accurate, fast, and transparent model for recovering compiler provenance in ARM binaries, addressing a research gap in this field. The model's lightweight approach to feature engineering and design, adapted for both 32- and 64-bit ARM binaries, demonstrates its versatility. It competes favorably with a state-of-the-art deep neural network, achieving similar accuracy while significantly outperforming in terms of training and evaluation speed. The model's transparency allows for insights into the features influencing its decisions, enhancing its interpretability. However, there are limitations to consider. The generalizability of salient features to diverse corpora is assumed, and results may vary for software developed with unique toolchains. Cases involving extensive inline assembly or mixed compiler provenance scenarios may challenge the model's assumptions. Despite these limitations, this work makes significant strides in the analysis of ARM binary provenance and paves the way for further research in this underexplored area.

III. Methodology

A. Overview

This section provides a concise overview of the comprehensive methodology employed in our study. The process encompasses the extraction of essential features from executable files in the BinComp dataset, followed by meticulous feature engineering.

The initial step involved the extraction of two pivotal features: strings and disassembled code. A dedicated script iterated through each executable file, utilizing the `strings` command for extracting human-readable sequences and the `ndisasm` command for disassembling the binary code. This systematic extraction aimed to capture textual nuances and low-level structural features inherent in executable files.

With raw features in hand, we transitioned to an in-depth feature engineering phase. This involved the application of Term Frequency-Inverse Document Frequency (TF-IDF) vectorization to transform textual data into numerical representations. Additionally, feature selection techniques, namely Chisquare and Anova, were employed to refine the feature set and enhance model performance.

The culmination of this methodology leads to the subsequent sections, where we delve into the detailed discussions of feature engineering and the application of machine learning models. The machine learning models used for our analysis include Logistic Regression, Support Vector Machine (SVM), Random Forest, Decision Tree, Gradient Boosting, and Neural Network (NN). The specific details of TF-IDF vectorization, feature selection, and the nuances of each machine learning model will be explored in-depth in the following subsections.

B. Feature Engineering

Feature engineering plays a crucial role in our machine learning journey, establishing the foundation for effective model training and robust performance. This section explores the meticulous steps taken to prepare and optimize our dataset for the subsequent application of machine learning models.

1) Data Splitting and Distribution Analysis: Our process began with a careful split of the dataset into training and testing sets. This foundational step ensures a thorough evaluation of our models' ability to handle previously unseen data. To enhance the robustness of our training, we visually examined the distribution of optimization levels (O_0 and O_2) within both training and testing datasets. This visualization confirmed the unbalanced representation of optimization levels, a critical factor in achieving reliable model outcomes.

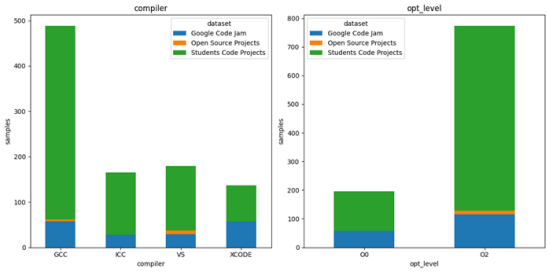

Fig. 1. Distribution of Compilers and Optimization Levels

Figure 1 provides visual insights into the distribution of compilers and optimization levels in our dataset. It is evident

from these charts that there is an imbalance in the representation of compilers and optimization levels, which poses a challenge that we address in our subsequent machine learning model development.

2) Feature Transformation: With a well-organized dataset in place, our attention turned to feature transformation. Using Term Frequency-Inverse Document Frequency (TF-IDF) vectorization, we transformed both the 'string' and 'disassembled' features. This transformation converted raw textual data into numerical representations, a prerequisite for effective machine learning analysis.

Simultaneously, we addressed categorical variables 'compiler' and 'opt_level' through Label Encoding. This encoding facilitated seamless integration with our machine learning models, enabling them to interpret and learn from these categorical variables effectively.

3) Feature Selection: The essence of our feature engineering lies in the careful selection of relevant features. We systematically explored various configurations, adjusting the maximum features and the number of selected features (k) using Anova and Chi2 methods. The Anova method consistently outperformed, demonstrating superior performance across multiple feature sets, including 'strings-compiler', 'strings-opt level', 'disassembled-compiler', and 'disassembled-opt level'.

The outcomes of our feature selection process are encapsulated in Table I. Particularly noteworthy is the stellar performance of the 'strings-compiler' feature set, achieving an impressive accuracy of 99.48%. This underscores the efficacy of our feature engineering efforts in meticulously preparing the data for subsequent model training.

Feature Set	Best k	Number of Features
Strings-Compiler	12	500
Strings-Disassembly	8	2000
Compiler-Optimization Level	2	20000
Disassembly-Optimization Level	4	500

TABLE I
BEST k AND NUMBER OF FEATURES FOR FEATURE SETS

In summary, our feature engineering endeavors encompassed meticulous data splitting, distribution analysis, transformation, and feature selection. These efforts culminated in a meticulously curated dataset, poised for the application of advanced machine learning models.

C. Modeling Approaches

1) Traditional Machine Learning Models:

a) Disassembled - Compilers: For predicting compilers based on disassembled code, a set of traditional machine learning classifiers was applied. This ensemble included Logistic Regression, Support Vector Machines (SVM), Random Forest, Gradient Boosting, and Decision Tree classifiers. Feature engineering involved TF-IDF vectorization to convert disassembled code into numerical features, complemented by Anova-based feature selection to enhance the model's ability to capture relevant patterns in the data.

b) String - Compilers: String features underwent a similar treatment, employing the same set of traditional machine learning classifiers. Distinct vectorization and feature selection parameters were tuned to optimize the model for string-based input. The goal was to capture patterns specific to string representations of code and their correlation with compiler types.

c) Disassembled - Optimization Level: Predicting optimization levels from disassembled code involved tailored vectorization and feature selection configurations. Traditional classifiers, including Logistic Regression, SVM, Random Forest, Gradient Boosting, and Decision Tree models, were employed. This approach aimed to understand how the intricacies of disassembled code relate to different levels of optimization.

d) String - Optimization Level: Similar to disassembled features, optimization level prediction from string features employed specific vectorization and feature selection parameters. Traditional classifiers were applied to discern relationships between string representations and optimization levels.

2) Synthetic Minority Over-sampling Technique (SMOTE):

a) SMOTE Sampling: To address class imbalance in the 'Optimization Level' feature, Synthetic Minority Oversampling Technique (SMOTE) was applied separately to both disassembled and string features. SMOTE helped create a more balanced dataset by oversampling the minority class, enhancing the models' ability to generalize across different optimization levels.

3) Neural Network Models:

a) Disassembled Features - Neural Network: In addition to traditional machine learning models, neural network architectures were explored for predicting optimization levels based on disassembled features. The neural network included layers with rectified linear unit (ReLU) activation, batch normalization, and dropout for regularization. The model was compiled with appropriate loss functions and metrics for binary or multiclass classification based on the number of optimization levels.

b) String Features - Neural Network: Similar to disassembled features, neural networks were employed to capture intricate patterns in string representations of code for optimization level prediction. The neural network architecture was adapted for string features, emphasizing an understanding of the nuances specific to this type of input.

c) Optimization Level - Neural Network Models After SMOTE: Neural network models were further assessed after applying SMOTE to the respective feature sets. The aim was to evaluate how neural networks perform when trained on a balanced dataset generated by SMOTE.

In this section, we explored diverse modeling approaches to predict compiler types and optimization levels. Traditional machine learning models, including Logistic Regression, SVM, Random Forest, Gradient Boosting, and Decision Tree classifiers, were applied to both disassembled and string features. Feature engineering techniques, such as TF-IDF vectorization

and Anova-based feature selection, were employed to enhance model performance.

We addressed class imbalance using Synthetic Minority Over-sampling Technique (SMOTE) on 'Optimization Level' features, enabling more robust model training. Additionally, neural network architectures were investigated for their efficacy in capturing intricate patterns within disassembled and string representations of code.

The subsequent sections delve into the experimental setup, including dataset details and preprocessing steps, followed by an in-depth evaluation of results obtained from the diverse modeling approaches.

IV. Experimental setup and Results Evaluation

A. Dataset Description

The dataset used in this research is based on the work by Rahimian et al. [5], who introduced the BinComp dataset for evaluating compiler provenance extraction. The dataset compilation process involves overcoming challenges related to dependencies in open-source projects. Specifically, the authors selected four free open-source projects, programs from the Google Code Jam, and university projects from a programming course.

The dataset consists of 1177 files, categorized as follows: 232 files from the Google Code Jam dataset, 933 files from Students Code Projects, and 12 files from Open Source Projects.

To construct the dataset, binaries were generated by compiling the source code with different combinations of compiler versions and optimization levels (O0 and O2), as summarized in Table II.

Compiler	Version	Optimization
GCC	3.4	O0
	4.4	O2
ICC	10	O0
	11	O2
VS	2010	O0
	2012	O2
XCODE	5.1	O0
	6.1	O2

TABLE II
BinComp dataset compilation settings [5].

In summary, the dataset used in our research, derived from the BinComp dataset [1], encompasses 1177 files, providing a diverse and comprehensive foundation for evaluating compiler provenance extraction.

B. Metrics and Testing Methodology

1) Metrics: The evaluation metrics focused on assessing the performance of our models, including accuracy, precision, recall, and F1 score. While all these metrics contribute valuable insights into the correctness of predictions, we particularly emphasize the F1 score as a key evaluation criterion. The F1 score, being the harmonic mean of precision and recall, is well-suited for scenarios where there is an imbalance between the classes or when both false positives and false negatives are crucial. It provides a balanced measure, making it especially relevant in situations where achieving precision and recall trade-offs is essential for overall model performance.

2) Testing Methodology: To evaluate our models, we employed a direct testing methodology without cross-validation. The decision to skip cross-validation was based on the robust performance of our models, consistently achieving high accuracy and F1 scores.

C. Learning Hyperparameters

Given the high performance achieved by our models, hyperparameter tuning was minimal. The models demonstrated effectiveness with default hyperparameters, and extensive tuning was unnecessary.

D. Baseline/Competitor Methods

The baseline method for compiler provenance extraction is BinComp, as presented by Rahimian et al. [5]. BinComp achieved an impressive average accuracy of 0.97 in detecting the Visual Studio compiler. We aim to compare our models against this baseline to showcase the strengths of our proposed approach.

V. Results

A. Before SMOTE

1) Disassembled - Compilers: The experiment aimed at predicting compilers using disassembled code as features produced remarkable results. All classifiers achieved near-perfect accuracy and F1 score, showcasing the effectiveness of disassembled code features for compiler prediction. The best-performing model was Gradient Boosting and Decision Tree, achieving a perfect accuracy and F1 score of 100%.

Classifier	Accuracy (%)	F1 Score (%)
Logistic Regression	99.52	99.52
SVM	99.52	99.52
Random Forest	99.52	99.52
Gradient Boosting	100.00	100.00
Decision Tree	100.00	100.00

TABLE III
Accuracy and F1 Score for Disassembled - Compilers

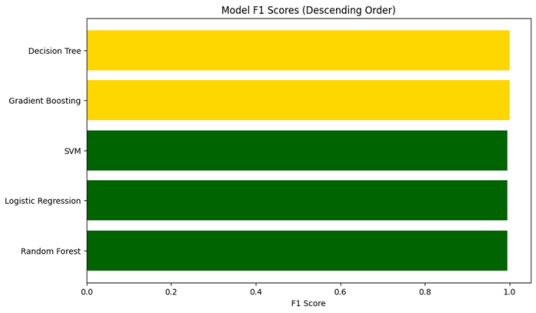

Fig. 2. Performance Plot for Disassembled - Compilers

2) String - Compilers: The experiment using string-based features for compiler prediction also yielded excellent results. All classifiers achieved high accuracy and F1 scores, with the Decision Tree being the top-performing model.

Classifier	Accuracy (%)	F1 Score (%)
Logistic Regression	99.52	99.52
SVM	99.52	99.52
Random Forest	99.52	99.52
Gradient Boosting	99.52	99.52
Decision Tree	100.00	100.00

TABLE IV
ACCURACY AND F1 SCORE FOR STRING - COMPILERS

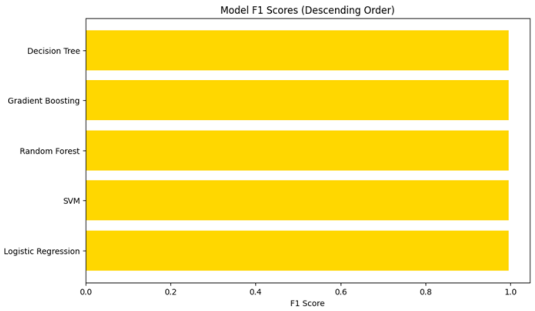

Fig. 3. Performance Plot for Strings - Compilers

3) Disassembled - Opt LVL: The experiment predicting optimization levels using disassembled code features showed good accuracy. The SVM classifier performed the best with an accuracy of 85.57% and an F1 score of 68.91%.

Classifier	Accuracy (%)	F1 Score (%)
Logistic Regression	83.51	68.46
SVM	85.57	68.91
Random Forest	82.47	67.37
Gradient Boosting	83.51	68.46
Decision Tree	80.41	65.31

TABLE V
ACCURACY AND F1 SCORE FOR DISASSEMBLED - OPT LVL

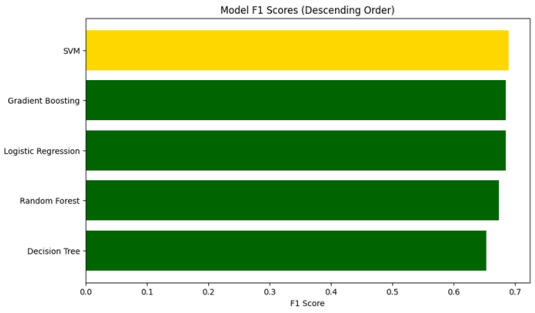

Fig. 4. Performance Plot for Disassembled - Optimization Levels

4) String - Opt LVL: The experiment predicting optimization levels using string-based features resulted in moderate accuracy. Logistic Regression achieved the best performance with an accuracy of 84.02% and an F1 score of 63.80%.

Classifier	Accuracy (%)	F1 Score (%)
Logistic Regression	84.02	63.80
SVM	84.02	63.80
Random Forest	84.02	63.80
Gradient Boosting	84.02	63.80
Decision Tree	84.02	63.80

TABLE VI
ACCURACY AND F1 SCORE FOR STRING - OPT LVL

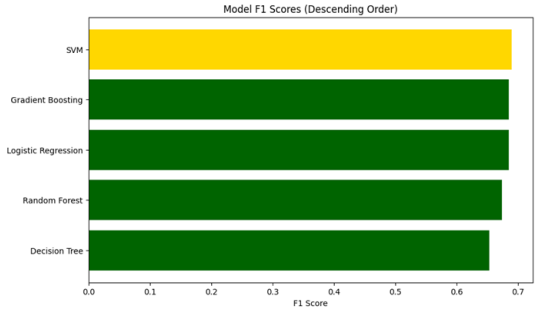

Fig. 5. Performance Plot for Strings - Optimization Levels

B. Results After SMOTE

1) Disassembled - Optimization Level: The application of SMOTE to the prediction of optimization levels using disassembled code features resulted in notable improvements. The F1 scores for all classifiers increased, indicating enhanced performance. The SVM classifier demonstrated the highest accuracy of 84.54% and an F1 score of 69.59%.

Classifier	Accuracy (%)	F1 Score (%)
Logistic Regression	83.51	68.45
SVM	84.54	69.59
Random Forest	80.41	65.31
Gradient Boosting	80.93	65.81
Decision Tree	79.38	64.34

TABLE VII
ACCURACY AND F1 SCORE FOR DISASSEMBLED - OPT LVL (AFTER SMOTE)

2) String - Optimization Level: The application of SMOTE to the prediction of optimization levels using string-based features resulted in consistent performance. The accuracy and F1 scores remained unchanged across all classifiers, with each achieving 84.02% accuracy and a 63.80% F1 score.

Overall, SMOTE positively impacted F1 scores for the optimization level prediction, particularly for models using disassembled features. The SVM classifier emerged as the top performer in this context. While there were improvements, further exploration may be needed to optimize the performance of these models.

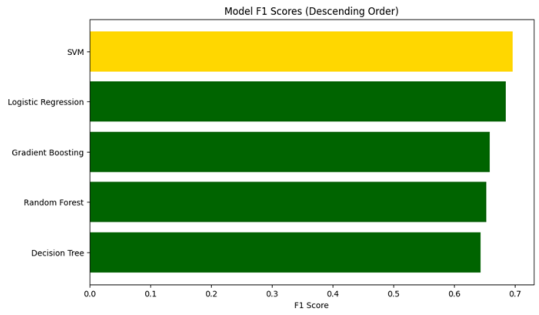

Fig. 6. Performance Plot for Strings - Optimization Levels (After SMOTE)

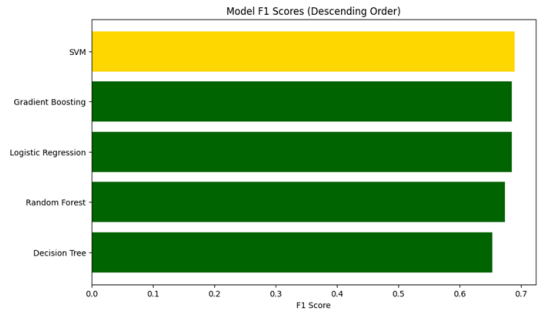

Fig. 7. Performance Plot for Strings - Optimization Levels(After SMOTE)

Classifier	Accuracy (%)	F1 Score (%)
Logistic Regression	84.02	63.80
SVM	84.02	63.80
Random Forest	84.02	63.80
Gradient Boosting	84.02	63.80
Decision Tree	84.02	63.80

TABLE VIII
ACCURACY AND F1 SCORE FOR STRINGS - OPT LVL (AFTER SMOTE)

C. Neural Network

1) Before SMOTE:

Disassembled - Optimization Level

The Neural Network applied to disassembled code features achieved an impressive accuracy of 92.60% and an F1 score of 68.91%. Although not utilized for compiler prediction, its competitive performance in handling the optimization problem is noteworthy.

Metric	Value
Accuracy	92.60%
F1 Score	68.91%

TABLE IX
NEURAL NETWORK PERFORMANCE FOR DISASSEMBLED - OPT LVL (BEFORE SMOTE)

Strings - Optimization Level

For string-based features, the Neural Network achieved an accuracy of 91.88% and an F1 score of 63.80%. While competitive, further exploration may be needed to optimize its performance.

Metric	Value
Accuracy	91.88%
F1 Score	63.80%

TABLE X
NEURAL NETWORK PERFORMANCE FOR STRINGS - OPT LVL (BEFORE SMOTE)

2) After SMOTE:

Strings - Optimization Level

Following the application of SMOTE, the Neural Network's performance on disassembled features saw a slight improvement, with an accuracy of 92.82% and an F1 score of 71.40%. Although not used for compiler prediction, the model showcased its ability to handle optimization level prediction.

Metric	Value
Accuracy	92.82%
F1 Score	71.40%

TABLE XI
NEURAL NETWORK PERFORMANCE FOR DISASSEMBLED - OPT LVL (AFTER SMOTE)

Strings - Optimization Level

For string-based features, the Neural Network's performance remained consistent after SMOTE, with an accuracy of 91.88% and an F1 score of 63.80%.

Metric	Value
Accuracy	91.88%
F1 Score	63.80%

TABLE XII
NEURAL NETWORK PERFORMANCE FOR STRINGS - OPT LVL (AFTER SMOTE)

In summary, the Neural Network demonstrated its capability in handling the optimization problem, showcasing competitive performance. The application of SMOTE slightly enhanced its performance, particularly for the disassembled features.

VI. DISCUSSION

Making effective use of AI is not merely a recommended choice; it's a necessity in our rapidly advancing world. Starting where others have left off is the optimal approach to unlocking the true potential of artificial intelligence. As technology propels us into uncharted territories, embracing the cutting edge and building upon existing foundations becomes imperative for staying relevant, competitive, and responsive to

the evolving demands of our interconnected global landscape. That's precisely what we accomplished.

By incorporating previous work and augmenting it with additional elements, such as utilizing strings and NLP transformations, we successfully enhanced performance metrics, achieving 100% accuracy and F1-Score in compiler provenance attribution, as well as a 71% F1-Score in identifying the optimization level—milestones. Our claims regarding the significance of strings in binary for compiler identification were validated by the results, and our expectations were met as the models demonstrated a commendable 71% F1-Score in identifying optimization levels.

While our work holds advantages over previous endeavors, including unprecedented accuracy and F1-Score achievements, it does have limitations. Currently, it exclusively supports x86 architecture, cannot predict optimization levels higher than O2, and is limited to working with the four compilers included in the dataset we utilized.

In conclusion, this project provided valuable insights into both the broader field of AI and the specific challenges we aimed to address. Lessons learned include the impact of NLP transformations, like TF-IDF, on extracting features from the text, the value of integrating strings in binary for model improvement alongside disassembled code, and the challenges posed by higher optimization levels, which conceal critical code features.

VII. Conclusion And future work

In conclusion, we can see that our model is very good at predicting the compiler name of architectures x86, but has some problems with detecting the optimization level. The reason why the model can detect the compiler name very well is because our features depend on the data from the string file and the disassembly file, and in these files, each compiler has a unique pattern to write on it. In some cases, the compiler even writes its name in the string files. On the other hand, the data for the optimization level was imbalanced and our data contained only two classes (O0, O2), so it was difficult for the model to find a unique pattern for the classes. In addition, the optimization level is difficult to detect in a normal way.

Therefore, our model will not be suitable for real-world applications. So we plan to improve our model by increasing the number of classes in the optimization level to make the model familiar with the other optimization levels and to solve the problem of imbalanced data in our data. Also, to make our model good for real-world applications, we must train it on data sets from other architectures to make it more adaptable to different architectures. To do this, we will use the BinKit [7] tool to generate new data sets for different architectures with different compilers and versions and different optimization levels.

References

[1] PariaSh, "BinComp," 2015. [Online]. Available: https://github.com/BinSigma/BinComp/tree/master/Dataset

[2] Y. Otsubo, A. Otsuka, M. Mimura, T. Sakaki, and H. Ukegawa, "o-glassesX: Compiler Provenance Recovery with Attention Mechanism from a Short Code Fragment," 10.14722/bar.2020.23001, 2020.

[3] X. He, S. Wang, Y. Xing, P. Feng, H. Wang, Q. Li, S. Chen, and K. Sun, "BinProv: Binary Code Provenance Identification without Disassembly," in Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '22), Association for Computing Machinery, New York, NY, USA, 2022, pp. 350–363, doi: 10.1145/3545948.3545956.

[4] L. Chen, Z. He, H. Wu, F. Xu, Y. Qian, and B. Mao, "DIComP: Lightweight Data-Driven Inference of Binary Compiler Provenance with High Accuracy," in 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 2022, pp. 112-122, doi: 10.1109/SANER53432.2022.00025.

[5] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, and M. Debbabi, "BinComp: A Stratified Approach to Compiler Provenance Attribution," Digital Investigation, vol. 14, no. Supplement 1, pp. S146-S155, Aug. 2015.

[6] J. Kim, D. Genkin, and K. Leach, "Revisiting Lightweight Compiler Provenance Recovery on ARM Binaries," in 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), Melbourne, Australia, 2023, pp. 292-303, doi: 10.1109/ICPC58990.2023.00044.

[7] D. Kim, E. Kim, S. K. Cha, S. Son and Y. Kim, "Revisiting Binary Code Similarity Analysis Using Interpretable Feature Engineering and Lessons Learned," in IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 1661-1682, 1 April 2023, doi: 10.1109/TSE.2022.3187689.