Pay Your Attention on Lib! Android Third-Party Library Detection via Feature Language Model

Dahan Pan*, Yi Xu*, Runhan Feng, Donghui Yu, Jiawen Chen, Ya Fang, Yuanyuan Zhang†
Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, China
Email: {dhpan98, yixu98, fengrunhan, yudhui, chenjiawen1353, sofo-s, yyjess}@sjtu.edu.com

***Abstract*—The widespread use of third-party libraries (TPL) has brought many conveniences to Android application development, fostering the development of the Android application ecosystem. Detecting the presence of TPLs in Android applications is crucial in the Android era, as it enables the rapid identification of their usage when security vulnerabilities arise in TPL code. Android code obfuscation can significantly impact the task of detecting TPLs, especially as obfuscation methods continue to iterate. Rule-based matching methods, which are commonly used in most approaches, often struggle to adapt to new obfuscation strategies.**

This paper proposes LibAttention, a feature-language-model-based Android TPL detection technique. LibAttention converts app binary code and TPL source code into Android intermediate representation Smali and extracts features that are less susceptible to obfuscation. These features are then fed into a language model to train an encoder from scratch. During the detection process, LibAttention encodes and compresses the app and TPL code representations and feeds them into downstream models for training and prediction.LibAttention is trained for third-party library detection tasks on downstream models, utilizing datasets compiled with various obfuscation modes and threshold adjustments to establish detection standards. Subsequently, detection and evaluation are conducted on the large-scale AndroZoo dataset. Its pretraining-fine-tuning model architecture eliminates the dependency on large amounts of labeled data samples.

The experimental results indicate that the detection capabilities of LibAttention are more effective compared to the baseline results, significantly mitigating the impact of the Android R8 obfuscation tool on applications. Moreover, when compared to existing rule-based Android TPL detection techniques, LibAttention demonstrates significant improvements on the Android R8 obfuscation dataset, boasting over a 30% enhancement in the F1-score.

***Index Terms*—Android, TPL detection, Feature Language Model, Transformer**

I. Introduction

To date, the Android application market thrives with the introduction of numerous new applications each year, contributing to its ever-growing ecosystem. The research findings indicate that in 2023, Google Play Store, one of the largest Android application markets, hosted 2.6 million applications with a total download count of 113 billion [1]. One of the reasons for the high frequency of new applications is the richness of TPLs available for Android, encompassing various additional functionalities such as advertising, location services, and payment components, which greatly facilitate app development for developers. As the Android software ecosystem evolves, TPLs become increasingly mature, leading to a higher frequency of their usage. However, due to inadequate understanding of TPL code, developers may sometimes overlook vulnerabilities present in such code when integrating it into their applications, thereby posing security risks and threatening user privacy. Previous studies have identified numerous vulnerabilities in TPLs used in Android applications (e.g., Facebook SDK) [2, 3], some of which may pose serious security risks, such as attackers exploiting vulnerabilities in certain advertising libraries (e.g., Airpush [4], MoPub [5]) to steal user privacy information. Compiled Android applications lack effective information indicating which TPLs they have used, making it difficult to identify potential risks posed by vulnerable TPLs. Consequently, the identification of TPLs has become an urgent task, classified under the broader software composition analysis (SCA) category.

Existing Android application TPL detection tools often employ rule-based methods, matching code features at the granularity of functions, modules, classes, etc., to identify TPLs used in obfuscated applications. Examples include LibScan [6], ATVHunter [2], PANGuard [7], which are based on similarity matching, as well as LibD [8], LibRadar [9], LibSift [10], which are based on clustering. These rule-based detection methods can only effectively detect applications using a few specific obfuscation mechanisms and cannot adapt to other obfuscation patterns, including Android R8. For instance, LibScan demonstrates outstanding performance on datasets obfuscated with ProGuard [11], DashO [12], and Allatori [13], accurately detecting the TPLs used in the applications. However, it fails to provide effective detection on datasets obfuscated with the Android R8 technique, as presented in our evaluation. Through manual analysis, we found that the detection rules of LibScan are unable to adapt to the obfuscation techniques used by Android R8, leading to a significant number of false negatives and false positives.

In summary, current Android application TPL detection tools suffer from two main shortcomings: 1) According to our evaluation, existing tools are ineffective in detecting applications obfuscated with Android R8, particularly struggling with changes brought about by code shrinking and optimization;

*These authors contributed equally to this work.
†Corresponding author.

2) Rule-based TPL detection tools lack robustness and are susceptible to changes in obfuscation rules, rendering them unsuitable for Android R8 obfuscation and potential future obfuscation methods.

To fill this gap, we designed and implemented LibAttention, the first feature-language-model-based technique capable of effectively detecting TPLs in Android applications using various obfuscation techniques. LibAttention operates through two main processes: model training and Android application TPL detection. During the model training process, LibAttention utilizes a pre-trained language model and trains an encoder from scratch using features extracted from a large corpus of applications obtained from the app market. This encoder is capable of extracting essential information from Android apps and TPLs. The detection process consists of three stages: data preprocessing, compression representation, and prediction. In the preprocessing stage, LibAttention converts the target application and TPLs into Android intermediate representation (Smali), extracts features, and encodes them using the pre-trained encoder. To further compress the information, LibAttention utilizes the properties of the pre-trained model to compress the encoder's output into a dense low-dimensional feature vector. Finally, these feature vectors are fed into downstream models to complete the training and prediction tasks.

In addition, the pretraining-fine-tuning model architecture used by LibAttention reduces the reliance on labeled samples with TPL tags. By autonomously learning Android code features from unlabeled data through self-regressive learning, and then learning TPL feature information from labeled data, LibAttention accomplishes training and prediction within a small sample space.

We summarize our contributions as follows:

1) We designed and implemented LibAttention, which is the first feature-language-model-based Android TPL detection technique. LibAttention extracts code features from a large number of unlabeled samples, which serve as the training data for the pretraining model. Subsequently, it utilizes the generated encoder and a custom compression algorithm to encode and compress the target app and TPL, producing dense low-dimensional feature vectors. Finally, these feature vectors are fed into downstream models for training and prediction.
2) We compared LibAttention with SOTA Android TPL detection techniques and our baseline methods on the Android R8 obfuscation dataset. The result shows that LibAttention demonstrates stable detection capabilities dataset. Moreover, we employ ablation studies to demonstrate that the deep learning approach integrated into LibAttention effectively neutralizes the impact of code obfuscation on third-party library detection.
3) We utilized LibAttention to detect vulnerable TPLs in a large-scale real-world scenario involving Android applications. The results indicate that among the 10,000 analyzed apps, 28 vulnerable TPLs were identified, with a total occurrence count of 42,408 times.

II. Background and Motivation

A. Background

In recent years, the Android app market has thrived, drawing increased attention to the security of its code. Software Composition Analysis (SCA) is an automated process that identifies open-source software components within a codebase. This analysis is conducted to assess security vulnerabilities, ensure license compliance, and evaluate code quality. Detecting third-party libraries (TPL) in Android application code is a subtopic of SCA.

Android code obfuscation is a crucial code protection measure within the Android development ecosystem. It diminishes code readability and enhances resistance to reverse engineering. Android code obfuscation primarily hardened the application from three perspectives: prevent reverse engineering, reduce the application size, and accelerate the execution of the program. Code obfuscation is widely used by app developers to prevent reverse engineering, making it harder for attackers to decompile and exploit vulnerabilities through existing analysis tools. Code-shrinking and resource-shrinking techniques can reduce the application size, which somehow reduces the program's attack surface. Code optimization involves altering the combination and sequence of program instructions, thereby enhancing the performance of the program during execution. Specifically, the Android R8 compiler handles 4 kinds of compile-time tasks:

- **Code Shrinking**. The compiler would detect and safely remove unused classes, fields, methods, and attributes from the application and its dependencies.
- **Resource Shrinking**. The compiler would remove the unused resources from the package application, including those in the app's dependencies.
- **Obfuscation**. The compiler would shorten the names of the Android application's classes, methods, and fields. The obfuscation task would not remove any code from the app or its dependencies.
- **Optimization**. The compiler would inspect and remove the redundant code at a deeper level, or possibly rewrite the code to make it less verbose.

B. Motivation

The Android obfuscation technique, originally designed to enhance Android code security, has become a significant impediment to third-party library detection, as it increases the complexity of the code and makes features more difficult to recognize. Although existing academic research has focused on specific obfuscation techniques with tailored solutions, these approaches often exhibit limited generalizability. The state-of-the-art Android TPL detection tool, LibScan [6], can effectively identify application libraries obfuscated with ProGuard, DashO, and Allatori, with an F1 score exceeding 97%. However, its detection capability sharply declines when it comes to applications obfuscated, shrunk, and optimized using Android R8. On the Android R8 obfuscated app dataset, the F1 score is only around 1%. This is because the strategies employed by Android R8 cannot be effectively accommodated by the detection rules of LibScan, leading to a high number of false negatives and false positives.

Through our analysis, we illustrated two scenarios that could lead to false negatives and false positives in SOTA TPL detection technique.

- **False Negative.** The R8 code optimization process uses inlining techniques, which alter the function call structure in the application. This change prevents the features extracted in the first step of SOTA technique from being effectively matched, resulting in false negatives.
- **False Positive.** Some simple functions in Android applications, such as get and set methods, have similar instruction sequences. This similarity can lead to errors when SOTA technique attempts to match based on instructions, resulting in false positives.

To make matters worse, the usage frequency of Android R8 obfuscation has been increasing. Android Studio, which is the Android officially presented development platform, uses Gradle to automate and manage the Android building process. When the developer builds the Android project using the Android Gradle plugin (AGP) 3.4.0 or higher version (currently 8.3), the plugin no longer uses ProGuard to perform compile-time code optimization but the Android R8 compiler instead. This implies that an increasing number of applications in the app market are being compiled and obfuscated using Android R8.

To address the aforementioned issues, one can employ deep learning-based methods, leveraging their ability to learn features from data. By doing so, it is possible to design a tool that can detect TPLs in applications obfuscated with Android R8, while also exhibiting generality and performing well across different obfuscation datasets. To achieve this, we face the following challenges in designing a deep-learning-based Android application TPL detection technique:

1) **Obfuscation Impact.** Android code obfuscation alters code features by changing variable names, control flow, and opcodes. Code shrinking may even remove variables and functions from the original program. These obfuscation outcomes increase the difficulty of Android TPL detection.
2) **Diverse Obfuscation Types.** Android code obfuscation techniques are diverse, including ProGuard, Allatori, DashO, and the increasingly popular Android R8. Each obfuscator has its own set of obfuscation rules, and new obfuscation tools and rules may emerge in the future. These factors contribute to the complexity of designing Android TPL detection tools, as they must be adaptable to various obfuscation techniques and anticipate future developments in code obfuscation.
3) **Reliance on Large Amount of Labeled Samples.** Deep learning methods often require learning features from a large amount of sample data to make predictions. However, obtaining labeled samples for Android TPL detection tasks often requires a certain amount of human effort. Relying on a large amount of labeled data would introduce a lot of additional workloads.

Therefore, we designed and implemented LibAttention, which is capable of detecting third-party libraries within Android applications across various obfuscation scenarios. Our experimental evaluation has demonstrated that LibAttention effectively mitigates the impact of code obfuscation on third-party library detection.

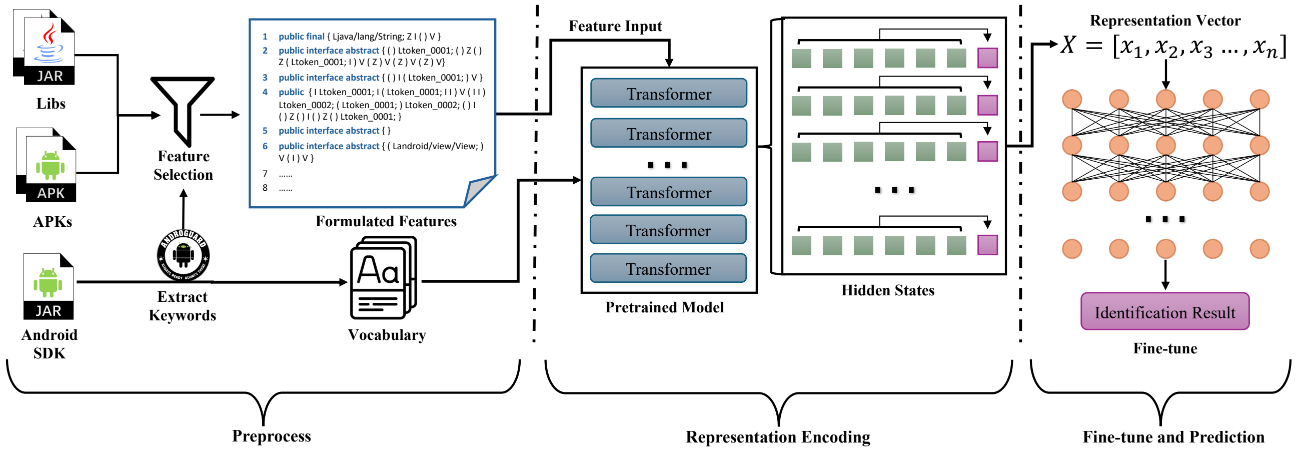
III. System Design

A. System Overview

The workflow of LibAttention is presented in Figure 1, which can be divided into three stages. The first stage involves data preprocessing. In this stage, LibAttention converts the original app and TPL code into Android's intermediate language Smali and extracts their features. Meanwhile, LibAttention establishes a vocabulary using keyword and type information from the Android SDK, which is utilized in the model's pre-training process and subsequent feature representation. The second stage comprises model pre-training and data representation encoding. The pre-training process constructs an encoder using a large amount of data as input through autoregressive training. Subsequently, utilizing this encoder and LibAttention's devised data compression scheme, the original data is compressed and represented. The third stage involves model fine-tuning and prediction. Using the output from the second stage as input, downstream models are constructed for specific tasks, and parameter fine-tuning is performed to complete the prediction task. We adopted a binary classification task for the model's fine-tuning training phase. By inputting the features of both the APK and the TPL, the model outputs a binary result of 0 or 1, which represents whether the APK contains the TPL.

LibAttention chose this technical approach for two main reasons. Firstly, Android applications are susceptible to various forms of obfuscation, where only a small portion of the information remains unaffected by obfuscation. Therefore, LibAttention retains only a small portion of the original code during data preprocessing. Even though the retained information may still be affected by obfuscation, this can be learned by the model in a heuristic manner. Secondly, the task of Android code TPL detection lacks rich labeled sample data in real-world scenarios. Training deep learning models on a small dataset often leads to overfitting. LibAttention addresses this issue by adopting a pretraining-fine-tuning approach. It learns the features of the code itself during the pre-training phase and then learns the information about the TPLs contained in the applications during the fine-tuning phase. This approach helps alleviate the dependency of deep learning on large sample data.

The detailed procedures of the three operational stages will be delineated in subsequent sections. Section III-B will elucidate the process of feature extraction and vocabulary construction. In Section III-C, the pre-training process will be expounded, encompassing model and data selection. Section III-D will delve into how LibAttention encapsulates code



```
public final { Ljava/lang/String; Z I ( ) V }
public interface abstract { ( ) Ltoken_0001; ( ) Z ( ) Z ( Ltoken_0001; I ) V ( Z ) V ( Z ) V ( Z ) V}
public interface abstract { ( ) I ( Ltoken_0001; ) V }
public { I Ltoken_0001; I ( Ltoken_0001; I I ) V ( I I ) Ltoken_0002; ( Ltoken_0001; ) Ltoken_0002; ( ) I ( ) Z ( ) I ( ) Z ( ) Ltoken_0001; }
public interface abstract { }
public interface abstract { ( Landroid/view/View; ) V ( I ) V }
......
......
```

Fig. 1. The overall workflow of LibAttention.

features. Lastly, Section III-E will outline the two modes of fine-tuning for the TPL detection task.

B. Preprocessing

The LibAttention is designed to extract the features that would persist during code shrinking, resource shrinking, code obfuscation, and optimization in the majority of cases. The extracted features comprise structural information of code classes, specific keyword occurrences within the code, and information regarding function and variable types immune to obfuscation effects. The extracted keyword information and type information will be combined to form a vocabulary, which will serve as the input for subsequent pre-training models. In our design, during the feature extraction process, the specific content of function bodies in the code is entirely removed, which is equivalent to a significant reduction in the original code. This approach is primarily motivated by two considerations: 1) Function body code is highly susceptible to the effects of code shrinking and code optimization during obfuscation, leading to inaccurate matching results. 2) Including function body code as input to machine learning models would rapidly expand the input space, thereby increasing the difficulty of model fitting. In addition to the removal of code function bodies, name information such as class names, method names, variable names, etc., which are highly susceptible to obfuscation, will also be removed during the preprocessing stage. Therefore, we only extract the aforementioned information from the original Android application as input features for downstream model training and TPL detection tasks. The specific content and processing workflow are outlined as follows.

Feature Selection and Vocabulary Construction. In this step, we need to extract data type information that will not be affected by obfuscation from both the original Android applications and library code.

Based on observation, we have found that during the Android application development process, only dependencies on specific Android SDK versions are specified. The SDK code itself is not packaged into the APK file but is instead sought from the execution environment during runtime. Therefore, all data types defined within the Android SDK will not be obfuscated. Otherwise, execution errors would occur due to name mismatch during runtime. We have selected the latest 10 versions of the Android SDK jar files and utilized the AndroGuard tool to extract all type information present within them as reserved types. The information extracted from the SDK includes Java basic types, Android definition types, and some special symbols. Theses information would be stored into the vocabulary.

Besides the type information extracted from the SDK, Android application code contains numerous custom types that are susceptible to obfuscation, including both application-specific types and types from library code. These types may

TABLE I
VOCABULARY LIST

Feature Level	Feature Type	Feature
Class Level Symbol	Keyword	public final abstract interface
Android SDK Reserved Symbol	Java Primitive Type	B S I J F D Z C V
	Parentheses	[( )

	Java Standard Type	begin with Ljava
		begin with Ljavax

	Android Standard Type	begin with Landroid
	Other Type	begin with Lorg
Obfuscated Symbol	Self Defined Type	Ltoken + ID

exhibit different forms after obfuscation. Directly adding the obfuscated results to the vocabulary would cause rapid inflation of the vocabulary, thereby prolonging the convergence time during model pre-training and failing to yield sufficient benefits for subsequent tasks. We have created a series of custom types to substitute for the types in the application code that may be obfuscated. We adopt a class-grained approach, replacing identical obfuscated types with the same custom type. This ensures the preservation of information regarding the use of the same type at different locations within the program. These custom types are also added to the vocabulary. Ultimately, the construction pattern of the vocabulary is as shown in Table I.

Feature Results Formulation To preserve the structural characteristics of the code to some extent, we extract the information from each class in the program as a row for model input X. We use additional "$\{\}$" symbols to denote the hierarchical relationship of functions, methods, and variables within a class. which is shown as:

$$X_p = \{c_1, c_2, \cdots, c_m\}\,\{\{m_1, m_2, \cdots, m_n\}\{v_1, v_2, \cdots, v_l\}\} \tag{1}$$

where every c_i refers to a class level symbol, v_k refers to a type variable, and m_j is a combination of parameter types, parentheses, and return value type, which is shown as:

$$m_j = ([parameters])\,[return] \tag{2}$$

Ultimately, the original Android applications and library files will be processed into the aforementioned format and fed into the model for training and prediction.

C. Pre-training

In this section, we have trained a pre-trained autoregressive language model using the previously extracted features to generate feature embeddings. The pre-trained model has two benign features. 1) The pre-trained model can learn code features from unlabeled samples, decreasing the reliance on labeled samples. 2) The pre-trained models can produce dense, low-dimensional feature embeddings, which further compress the code features without losing essential information.

Specifically, we pre-trained a GPT-like model from scratch, initializing the parameters randomly. Our rationale for selecting a GPT-like architecture, as referenced in [14], lies in its capacity as a versatile language model. Unlike BERT, which operates as a masked language model, GPT-like models offer flexibility during pre-training. BERT confines input to individual sentences, each constrained by a fixed token length limit, whereas GPT-like models allow for a more natural, continuous flow of text. In our pre-training dataset, the average length of each document surpasses 100K tokens, significantly exceeding the input threshold for BERT models. Conversely, GPT models are capable of processing lengthy text inputs without such limitations, rendering them more apt for handling our pre-training data. To facilitate autoregressive pre-training, we employed a meticulously curated vocabulary and dataset. The autoregressive loss function $\mathcal{L}$ is as follows:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\log P(y_{i,t}|y_{i,<t}, \theta) \tag{3}$$

where N is the number of samples, T is the sequence length, $y_{i,t}$ represents the true label at position t of the ith sample, $y_{i,<t}$ denotes the sequence of labels up to position t of the ith sample, $P(y_{i,t}|y_{i,<t}, \theta)$ is the probability of the label at position t given the sequence of labels up to position t of the ith sample parameterized by θ, and θ represents the model parameters. By iteratively learning from the data through multiple rounds, the loss function $\mathcal{L}$ gradually decreases, thus completing the pre-training process.

We selected over 1,500 Android application APKs from the public dataset AndroZoo, along with a batch of popular TPL data downloaded from Maven, as the pre-training data for training. In total, this dataset comprises over 200 million tokens. The objective is to train an encoder capable of learning code features.

D. Representation Encoding

Generally, the output embeddings of pre-trained transformers can be directly utilized as input for downstream task-specific models. However, in the task of detecting Android TPL code, the input data is scaled at the granularity of APKs. Even after encoding and compression through transformers, the embedding volume remains substantial, making it impractical to directly feed into downstream models. Therefore, further compression of the data is required.

Algorithm 1: Representation Encoding Algorithm

```
Input: The preprocessed APK or JAR feature $X$
       with $L \times d$ dimension; model.pth
Output: The encoding of input file $Y$ with $1 \times d$
        dimension
1 Initialize Transformer $T$ = model.pth;
2 Initialize hidden_states $hs_0$ = $X$;
3 compressed_hidden_states = [];
4 for hidden_layer $l_i$ in Transformer $T$ do
5     hidden_states $hs_i$ = calculate($l_i$, $hs_{i-1}$);
6     represent_state $rs_i$ = $hs_i$[represent_id, :];
      /* Replace L-dimensional data
         with 1-dimensional data       */
7     compressed_hidden_states.append($rs_i$);
8 end
9 $Y$ = compressed_hidden_states[selected_layer];
```

We have devised a method to further compress embedding vectors, reducing data dimensions while retaining the main data features to make sample features acceptable to downstream models. Specifically, since we selected the GPT-like model for pre-training, each symbol in the generated embedding vectors contains partial information from other symbols with the help of the attention mechanism. Therefore, we can select a subset of symbols to represent the overall features. The specific algorithm is shown in Algorithm 1.

After completing pre-training, we obtained a pre-trained model *model.pth* consisting of several hidden layers. For each pre-processed data instance, it is fed into the pre-trained model for encoding. We obtain the hidden states of the data at each layer l_i and select the represent state rs_i to represent the entire hidden states. Finally, we choose the hidden states from one layer as the representation of the entire data. The selection of hidden states depends on the predictive performance of the downstream model.

After experimenting with different hidden states and hidden layers, we found that selecting the last state of each layer, along with the last hidden layer, provided the optimal detection performance. We hypothesize that this is because the output of GPT-like models often contains partial information from previous states. Thus, selecting the last state allows for data compression with minimal information loss.

E. Fine-tuning

During the fine-tuning phase, we leveraged labeled data to adjust the parameters of the final three layers of the Transformer architecture. We use the pre-training stage to enable the model to recognize features of Android applications and TPL code. Then, through the fine-tuning stage, we adapt the model to the specific task of detecting TPLs in Android applications. We employed the most conventional strategy for third-party library detection tasks as our downstream task. Specifically, the features of both the APK and the TPL are input into the system, which then outputs a label indicating whether the APK contains the TPL. This presence or absence is represented by binary labels, 0 or 1.

In terms of parameter selection, we utilized Binary Cross-Entropy Loss (BCE loss) to adjust the model parameters during the training process. BCE loss is a commonly used loss function for binary classification tasks. The formula for BCE loss $\mathcal{L}_{BCE}$ is as follows:

$$\mathcal{L}_{BCE} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot \log(p_i) + (1-y_i)\cdot\log(1-p_i)] \quad (4)$$

Here, N is the number of samples, y_i represents the true label (0 or 1) for each sample, and p_i is the predicted probability of the sample being in class 1. This loss function is particularly effective for binary classification tasks as it quantifies the difference between the predicted probabilities and the actual binary outcomes, thereby guiding the model towards more accurate predictions.

For the activation function, we used the Sigmoid function, which is commonly applied in binary classification tasks to map predictions to probabilities between 0 and 1. The formula for the Sigmoid function is as follows:

$$S(x) = \frac{1}{1+e^{-x}} \quad (5)$$

TABLE II
DATASET SETTINGS

Dataset	Category	#apps	Have GT	Release Source
$AD_1 \| \| LD_1$	R8-non	49	Yes	build from source code
	R8-obf-shr-opt	49		
	R8-obf-shr-rcs	48		
	R8-obf-shr-rcs-opt	48		
AD_2	Mixed	1,500	No	Google Play
$AD_3 \| \| LD_3$	Mixed	10,000	No	AndroZoo Maven

In addition to the initial approach, we also experimented with using a Multi-Layer Perceptron (MLP) to replace the Transformer layers during the fine-tuning phase for downstream training. We observed that the MLP was able to fit the data more quickly than the Transformer. However, there was no significant difference in prediction performance between the two models. This suggests that while MLPs may offer computational efficiency, they do not necessarily provide a superior predictive capability over Transformers in this specific application.

IV. Evaluation

This section evaluates the effectiveness and efficiency of LibAttention. Also, we use LibAttention to detect vulnerable TPLs in real-world Android apps.

A. Experimental Setup

1) ***Experimental Environment Configuration:*** Our experiments are conducted on two servers: one for data acquisition and preprocessing, equipped with Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz, 128GB RAM, and running Ubuntu 20.04 operating system. The other one for model pre-training, data encoding, and fine-tuning, including an Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz, a GeForce RTX3090 GPU with 24GB VRAM, 128GB RAM, and running Ubuntu 20.04 operating system with CUDA 12.0. We develop LibAttention with *Python* 3.9. LibAttention extracts the features from apps with *AndroGuard* 3.3.5 [15]. For TPLs, LibAttention compiles the jar or aar files with *dx* to convert them into Dex files, then extracts the features with *AndroGuard*.

2) ***Dataset Construction:*** To construct our model and rigorously evaluate our methods, we designed three distinct application datasets, labeled $AD_1 \sim AD_3$, along with two correlated library datasets, LD_1 and LD_3, as detailed in Table II.

The applications in AD_1 were meticulously selected at random from the FDroid open-source dataset, which is known for its diverse range of Android applications. These applications were then manually compiled using four distinct obfuscation schemes facilitated by the latest version of Android Studio Hedge, 2023.1.1, to simulate various real-world software obfuscation scenarios that might be encountered in an adversarial environment. The corresponding dataset of third-party libraries, LD_1, was meticulously assembled by extracting library data from the configuration files of these applications, ensuring a comprehensive capture of library dependencies and usage. AD_2 consists of 1,500 Android applications, all released after the year 2023, and sourced from the Google Play store. These applications were subsequently downloaded from the AndroZoo [16] dataset, which is frequently updated and widely recognized for its vast repository of live app data, reflecting contemporary market trends. This dataset is strategically chosen to serve as a benchmark for the real-world applicability of our proposed LibAttention model by providing a dynamic and up-to-date representation of the current Android application landscape. AD_3 contains a larger set of 10,000 Android applications, also sourced post-2023 from AndroZoo, offering an extensive base for validating the scalability and robustness of our model under varied conditions. Lastly, LD_3 includes an collection of 344 different versions of 52 third-party libraries (TPLs) that incorporate CVE tags, sourced from the most popular pages of the Maven [17] Android library repository. This dataset is crucial for assessing the vulnerability detection capabilities of our model, as it comprises a wide range of common and critical libraries used across numerous applications.

*3) **Comparison Technique:*** For our comparison experiments, we established a baseline configuration that corresponds to the LibAttention scheme called LibAttention^I, along with three Android third-party library detection techniques widely used in both academic and industry settings: LibScan [18], LibID [19], and LibScout [20]. This setup aims to evaluate the effectiveness and robustness of LibAttention against established methodologies in the domain of Android third-party library detection.

Specifically, as shown in 2, LibAttention^I retains the data preprocessing procedures of LibAttention. Subsequently, the extracted features are transformed into SimHashes based on the vocabulary of LibAttention. The presence of Android third-party libraries within applications is then determined by whether the proportion of matching SimHash values exceeds a predefined threshold θ [21].

During this process, employing the SimHash algorithm can mitigate the impact of discrepancies in symbol declaration order caused by differences in the compilers used for APKs and TPLs on detection results. Experimental findings demonstrate that the use of SimHash can slightly enhance detection precision.

*4) **Evaluation Metrics:*** We employ the F1 score as a key metric to measure the effectiveness. The F1 score, which harmoniously balances precision and recall, is particularly effective in scenarios where an equal importance is assigned to both false positives and false negatives. This metric is crucial for evaluating the accuracy and reliability of our detection methods in identifying the presence of third-party libraries in Android applications.

$$Precision = \frac{TP}{TP + FP} \quad Recall = \frac{TP}{TP + FN}$$

Algorithm 2: LibAttention^I Algorithm

```
1:  Input: Extracted APK features $apk\_feature\_ls$,
    Extracted TPL features $tpl\_feature\_ls$, Threshold $\theta$,
    vocabulary $v$
2:  Output: Boolean result $is\_contained$
    {Convert APK features to and TPL features SimHash
    values}
3:  $apk\_sim\_hash\_ls \leftarrow SimHash(apk\_features\_ls)$
4:  $tpl\_sim\_hash\_ls \leftarrow SimHash(tpl\_features\_ls)$
    {Check for containment using a threshold}
5:  Initialize match count: $inside \leftarrow 0$
6:  Initialize total comparisons:
    $total \leftarrow len(tpl\_sim\_hash\_ls)$
7:  for each $tpl\_sim\_hash$ in $tpl\_sim\_hash\_ls$ do
8:     if $tpl\_sim\_hash \in apk\_sim\_hash\_ls$ then
9:        $inside \leftarrow inside + 1$
10:    end if
11: end for
12: if $inside/total > \theta$ then
13:    return true
14: end if
15: return false
```

$$F1\text{-}score = \frac{2 \times Recall \times Precision}{Recall + Precision}$$

The *true positive* (TP) refers to the case that the app contains a TPL, and the tool detects the existence of this TPL. The *false positive* (FP) means the tool reports some TPL that does not exist in the app. The *false negative* (FN) represents that a TPL contained in an app is reported absent by the tool.

*5) **Research Questions:*** In the Evaluation section, we will address the following three research questions:

- RQ1: How does LibAttention compare in effectiveness to existing third-party library detection tools?
- RQ2: Can LibAttention's deep learning-based approach effectively mitigate the impact of code obfuscation on third-party library detection?
- RQ3: What is the detection efficiency of LibAttention ?

B. Effectiveness Evaluation

*1) **Effectiveness Comparison with State-of-the-art TPL Detectors:*** We compare the effectiveness of LibAttention with LibID [19], LibScout [20], LibScan [18] and our baseline LibAttention^I. We conducted our performance comparison experiments using the dataset comprising AD_1 and LD_1. This dataset was selected to facilitate a comprehensive evaluation of our model's effectiveness in detecting third-party libraries within Android applications. Specifically, we selected 60% of the APKs from the dataset, along with their corresponding third-party library data, for model training and threshold adjustment. All comparative results were then obtained using the remaining 40% of the dataset, ensuring a robust evaluation of the model's performance. The experiment results are presented in Table III. In Table III, we use "non" to indicate applications

TABLE III
EFFECTIVENESS COMPARISON OF DIFFERENT TOOLS ON $AD_1||LD_1$

Obfuscation Level	LibAttention			LibAttention^I			LIbScan			LibScout			LibID-S		
	Pre	Rec	F1	Pre	Rec	F1	Pre	Rec	F1	Pre	Rec	F1	Pre	Rec	F1
R8-non	0.391	0.379	0.385	0.209	0.791	0.331	0.169	0.085	0.113	0.302	0.849	**0.446**	0.266	0.703	0.386
R8-obf-shr-opt	0.402	0.379	**0.390**	0.102	0.127	0.113	0.010	0.002	0.003	0.022	0.005	0.008	0.049	0.019	0.027
R8-obf-shr-rcs	0.386	0.379	**0.383**	0.232	0.493	0.316	0.010	0.003	0.005	0.023	0.016	0.019	0.160	0.151	0.155
R8-obf-shr-rcs-opt	0.396	0.379	**0.388**	0.101	0.127	0.112	0.000	0.000	0.000	0.009	0.004	0.006	0.041	0.010	0.016

that have not been obfuscated, "obf" for those obfuscated with R8, "shr" for applications subjected to code shrinking, "rcs" for those undergoing resource shrinking, and "opt" for apps optimized through R8 compilation.

The results in Table III clearly indicate that LibAttention exhibits significantly superior performance compared to other detection tools and the baseline across three different obfuscation modes. The only exception is in the case of the non-obfuscated dataset, where its performance is slightly lower than that of LibScout.

Considering that previous studies have highlighted the notable advantages of LibScan in detection performance, its poor performance on the $AD_1||LD_1$ dataset prompted us to conduct a thorough manual analysis to uncover the underlying reasons for this discrepancy. We found that, in order to improve matching effectiveness, LibScan conducts class-level filtering in the signature-based class correspondence detection step, where only classes strictly matching LibScan's rules are considered as candidate classes and proceed to subsequent filtering steps. Due to the impact of R8 on the features used by LibScan during obfuscation optimization and code reduction stages, there are occasions where certain functions or variables within classes are deleted, leading to inaccurate matching results.

On the contrary, LibAttention utilizes coarse-grained features of Android applications, which are less susceptible to obfuscation. Moreover, LibAttention employs deep learning techniques, enabling it to learn obfuscation patterns from data, thereby effectively resisting the impact of different obfuscation techniques on detection results. (RQ1)

2) ***LibAttention's robustness on Different Obfuscation Levels:*** In this section, we primarily discuss the comparative results between LibAttention and our designed baseline, LibAttention^I, to illustrate the effectiveness of the feature-language model we employed, as well as the downstream training model. This analysis underscores the advantages of our approach in detecting third-party libraries in Android applications.

First, we provide an explanation of the experimental results presented for the baseline in Table III. These results are derived from a unified decision threshold applied across the four different obfuscation modes. While utilizing separate thresholds for each obfuscation mode could yield improved F1 scores, such an approach would not be fair and would fail to reflect real-world scenarios accurately. Under this circumstance, it becomes evident that the experimental results for the baseline are highly susceptible to the effects of Android obfuscation. Even though we employed a SimHash strategy during the detection process(, which provided some improvement compared to methods that do not utilize SimHash), the performance still significantly deteriorates when applications are subjected to R8 optimization strategies, particularly Method Inlining.

The results in Table III indicate that LibAttention is almost unaffected by obfuscation, with its detection performance closely approaching the optimal results achieved by LibAttention^I on non-obfuscated data. We believe this is due to LibAttention's ability to learn the impact of Android obfuscation strategies on data features during the training process, effectively mitigating their interference with detection outcomes. This also demonstrates the effectiveness of our proposed feature set and the training plus downstream task detection paradigm. (RQ2)

C. Performance on Large-scale Dataset

This section focuses on evaluating the performance of LibAttention using the $AD_3||LD_3$ dataset, sourced from AndroZoo and Maven, which closely resemble real-world scenarios. We measure the performance of LibAttention in three stages: preprocessing, representation encoding, and prediction, denoted as T1, T2, and T3 respectively. Four metrics (`Q1`, `Median`, `Q3`, and `Average`) are employed to assess the time expenditure of LibAttention. `Q1`, `Median`, and `Q3` represent the first quartile, median, and third quartile, respectively, obtained by sorting the detection time expenditure of LibAttention from low to high. `Average` denotes the average time spent on detection on the $AD_3||LD_3$ dataset. The per-app time costs on $AD_3||LD_3$ are presented in Table IV.

TABLE IV
PER-APP EFFICIENCY ON DIFFERENT LIBATTENTION STEPS

	Token Number	T1(s)	T2(s)	T3(s)	Total(s)
`Q1`	110,712	103.3	1.1	<0.1	104.4
`Median`	252,031	516.5	2.5	<0.1	519.0
`Q3`	300,630	752.5	3.0	<0.1	755.5
`Average`	201,922	470.1	2.2	<0.1	472.1

The results from the table indicate that the primary time expenditure of LibAttention stems from the data preprocessing stage, where LibAttention utilizes AndroGuard to extract code features from both the app and TPL. The encoding representation process can handle approximately 100,000 tokens per second on average, with an average encoding time of about 2.2 seconds per app. The prediction process consumes less time on average per app, with the total time for 10,000 applications and 285 TPL matches being less than one hour. Fortunately, in practical detection scenarios, parallel algorithms can be employed by running multiple processes simultaneously to reduce time expenditure. In the actual detection process of LibAttention, we utilized 40 processes for experimentation, keeping the overall experiment time within two days.

Also, we employ LibAttention to conduct large-scale detection of vulnerable TPLs in real-world Android applications. We obtained all TPLs containing CVEs from the popular TPL list on Maven, forming the vulnerable TPL dataset LD_3. Subsequently, we perform detection on a dataset comprising 10,000 apps downloaded from AndroZoo after 2023. The detection results are illustrated in Figure 2.

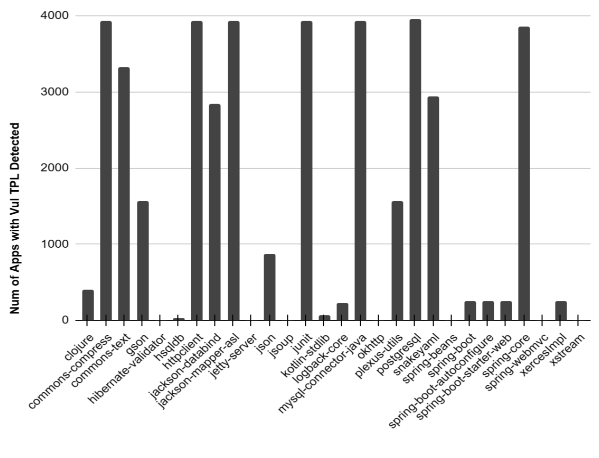


Fig. 2. Large-Scale Vulnerable TPL Detection Results.

The detection results reveal that among the 10,000 apps analyzed, 28 vulnerable TPLs were utilized, with a total occurrence count of 42,408 times. Further analysis on specific vulnerability types, potential risks, and other related information will be conducted as part of our future work.

V. Discussion

In this section, we will discuss three key aspects: the rationale behind choosing deep learning models for third-party library detection, our experimental results, and the future directions for the development of third-party library detection tools. This comprehensive discussion aims to highlight the strengths of our approach and envision potential advancements in this field.

First, we discuss the motivation for incorporating deep learning models. In recent years, numerous papers on Android third-party library detection have been published, each demonstrating the superiority of their respective tools through experimental validation. However, the rule-based systems commonly used in these tools are susceptible to obfuscation strategies, resulting in inconsistent performance across different datasets. Therefore, we propose using deep learning to automatically learn code features, aiming to neutralize the effects of code obfuscation in the task of third-party library detection. This approach enables our model to adapt and generalize better across various obfuscation techniques and datasets.

Next, we delve into a detailed discussion of the experimental results presented in Section IV-B. Table III shows that the highest F1 scores achieved by all tools on our dataset do not exceed 40%, which is relatively low compared to results reported in other studies. Upon further analysis of the data, we hypothesize that this lower performance is due to our non-selective approach in choosing third-party libraries for the dataset, resulting in high variance in library features. We believe that this scenario is more reflective of real-world applications, and thus, we have opted to retain the original experimental outcomes in our presentation. This decision underscores our commitment to providing realistic and practical insights into third-party library detection under varied and challenging conditions.

Finally, we discuss the future directions for third-party library detection methods. Based on our review of existing work and the conclusions drawn from our experiments, we foresee two potential developmental pathways for third-party library detection:

1) We believe that to design a tool capable of handling all obfuscation strategies, it is imperative to incorporate deep learning technologies, supported by extensive datasets and larger model for training. This approach will allow the detection tool to learn and adapt to a wide array of obfuscation patterns, enhancing its robustness and reliability.

2) In the absence of large-scale data, it may be feasible to decompose the task of third-party library detection into two sub-tasks: detecting obfuscation patterns and then detecting third-party libraries under specific obfuscation conditions. This modular approach could simplify the challenges associated with library detection in obfuscated code.

Moving forward, we plan to design and implement such solutions, aiming to bridge the gaps identified in current methodologies and push the boundaries of what can be achieved in third-party library detection within the Android ecosystem.

VI. Related Work

In recent years, research on software composition analysis (SCA) has primarily focused on two approaches: rule-based detection [2, 3, 6, 8, 22–28] and deep-learning-based detection [29].

The rule-based detection method refers to computing the similarity between detection objects by extracting preserved features (such as features unaffected by obfuscation or optimization) from both the detection source and target. The matching result is determined based on a similarity threshold. LibDetect [28] extracts five abstract code features, including bytecode, identifiers, control flow, and other features. It also uses fuzzy hashing of Android application code to mitigate the effects of obfuscation. LibScout [20, 27] utilizes class hierarchy to profile the features of Android TPLs and set up a library database. It flattens the package layer by generating a fixed-depth Merkle tree and generates the library signatures. The signatures are finally used for comparison. Orlis [26] generates call graph signatures for both Android applications and TPLs and compares them using digest-based similarity scores. LibPecker [3] constructs a strict class signature based on class dependency relationships. It then completes the comparison between Android applications and TPLs based on the similarity of these class signatures. LibID [19, 25] consists of two detection schemes: LibID-S and LibID-A, focusing on scalability and accuracy, respectively. This tool overcomes some limitations in previous work by addressing techniques such as identifier renaming, code shrinking, control flow randomization, and package modification. It can also determine the versions of TPLs used in the application binary. PANGuard [7] and ATVHunter [2] are similar in that they both decouple TPL code from application code based on static information such as Program Dependence Graphs (PDGs) or Control Dependence Graphs (CDGs) in Android applications. PANGuard utilizes both structural and content information as features and employs a feature set matching algorithm to identify TPLs in applications. ATVHunter, on the other hand, employs coarse-grained and fine-grained approaches. In the coarse-grained phase, it assigns a unique serial number to each basic block within a method and transforms the intra-procedural Control Flow Graphs (CFG) from the adjacency list into a method signature based on assigned serial numbers. In the fine-grained phase, it uses fuzzy hashing on each opcode within a sliding window operation to avoid significant differences in the final fingerprint caused by local feature changes triggered by obfuscation. OSSFP [22] operates at the function level, preserving core functions within the application to construct a fingerprint index for each TPL project. It effectively eliminates noise interference in the fingerprint, achieving both high performance and high accuracy. LibScan [6, 18] extracts features from Android applications and TPL code at the class level. It performs feature matching between applications and libraries from three perspectives: class-signature correspondence, method-opcode similarity, and call-chain-opcode similarity. LibScan achieves high accuracy even on applications obfuscated using traditional methods such as ProGuard, DashO, and Allatori, while also demonstrating high detection performance. These rule-based detection methods share a common limitation: they can only effectively detect applications subjected to specific obfuscation patterns. Any changes in the obfuscation methods can significantly impact the effectiveness of these techniques, leading to additional false negatives and false positives. Particularly, in recent years, the majority of Android applications have adopted the Android D8/R8 obfuscation mode, which these methods cannot adequately support.

The deep-learning-based SCA method refers to a technique where a tool utilizes a deep learning model to extract and learn features from existing data, enabling it to possess certain recognition capabilities. Some models with code clone detection capabilities are able to perform SCA tasks, including binary code clone detection [30–34], source code clone detection [35–42], and binary-to-source code detection [29]. In particular, as a binary-to-source code clone detection tool, BinaryAI [29] focuses on SCA detection. It utilizes the Pythia [43] and then further performs pre-training using contrastive learning, enhancing data features to achieve feature matching between application code and decompiled pseudo-code of TPLs. However, these tools can only perform comparison work at the granularity of functions and cannot directly compare entire libraries or entire application code. Even though BinaryAI has demonstrated its ability to perform SCA at the function level in C/C++ programs, this approach has not yet been tested with Android obfuscation methods.

LibAttention combines the advantages of two types of methods. It inherits the characteristics of non-heuristic similarity-based matching methods, which operate at the granularity of classes or modules. Additionally, it learns from the advantages of heuristic methods utilizing deep learning models during the matching process. This enables it to detect code libraries contained within Android applications obfuscated using different obfuscation methods. Particularly, it can accurately perform SCA tasks, even under the widely used Android R8 obfuscation mode in recent years.

VII. Conclusion

The proposed system, LibAttention, employs a feature-language-model-based approach for Android third-party library (TPL) detection. It begins by converting app binary code and TPL source code into Android's intermediate representation Smali, extracting features less prone to obfuscation. These features are then utilized to train an encoder from scratch using a language model. In the detection phase, LibAttention encodes and compresses the app and TPL code representations and employs two downstream models for training and prediction, tailored for different detection scenarios. Furthermore, compared to existing rule-based Android TPL detection techniques, LibAttention exhibits significant improvements on the Android R8 obfuscation dataset, boasting over a 30% enhancement in the F1-score.

References

[1] David Curry, "Android statistics (2024)," 2024, https://www.businessofapps.com/data/android-statistics/, Last accessed on 2024-03-19.

[2] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1695–1707.

[3] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in android

applications with high precision and recall," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 141–152.

[4] "Airpush," https://support.google.com/faqs/answer/6376-737.

[5] "Mopub," https://support.google.com/faqs/answer/63459-28.

[6] Y. Wu, C. Sun, D. Zeng, G. Tan, S. Ma, and P. Wang, "Libscan: towards more precise third-party library identification for android applications," ser. SEC '23. USA: USENIX Association, 2023.

[7] Z. Tang, M. Xue, G. Meng, C. Ying, Y. Liu, J. He, H. Zhu, and Y. Liu, "Securing android applications via edge assistant third-party library detection," *Computers & Security*, vol. 80, pp. 257–272, 2019.

[8] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in android markets," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 335–346.

[9] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 653–656.

[10] C. Soh, H. B. Kuan Tan, Y. L. Arnatovich, A. Narayanan, and L. Wang, "Libsift: Automated detection of third-party libraries in android applications," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, 2016, pp. 41–48.

[11] "Guardsquare. proguard," https://www.guardsquare.com/proguard.

[12] "Preemptive. dasho," https://www.preemptive.com/products/dasho.

[13] "Smardec inc. allatori," http://www.allatori.com.

[14] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[15] "Androguard," https://androguard.readthedocs.io/en/v3.3.5/intro/gettingstarted.html.

[16] "Androzoo," https://androzoo.uni.lu/.

[17] "Maven repository: Search/browse/explore," https://mvnrepository.com/.

[18] Y. Wu. Libscan: towards more precise third-party library identification for android applications. [Online]. Available: https://github.com/wyf295/LibScan

[19] J. Zhang. Libid: reliable identification of obfuscated third-party android libraries. [Online]. Available: https://github.com/ucam-cl-dtg/LibID

[20] M. Backes. Libscout. [Online]. Available: https://github.com/reddr/LibScout

[21] R. Feng, Z. Zhang, Y. Zhou, Z. Yan, and Y. Zhang, "Accurate and efficient code matching across android application versions against obfuscation," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2024, pp. 204–215.

[22] J. Wu, Z. Xu, W. Tang, L. Zhang, Y. Wu, C. Liu, K. Sun, L. Zhao, and Y. Liu, "Ossfp: Precise and scalable c/c++ third-party library detection using fingerprinting functions," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 270–282.

[23] W. Tang, Y. Wang, H. Zhang, S. Han, P. Luo, and D. Zhang, "Libdb: an effective and efficient framework for detecting third-party libraries in binaries," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 423–434.

[24] B. Li, Y. Zhang, J. Li, R. Feng, and D. Gu, "Appcommune: Automated third-party libraries de-duplicating and updating for android apps," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 344–354.

[25] J. Zhang, A. R. Beresford, and S. A. Kollmann, "Libid: reliable identification of obfuscated third-party android libraries," ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 55–65.

[26] Y. Wang, H. Wu, H. Zhang, and A. Rountev, "Orlis: obfuscation-resilient library detection for android," ser. MOBILESoft '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 13–23. [Online]. Available: https://doi.org/10.1145/3197231.3197248

[27] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 356–367.

[28] L. Glanz, S. Amann, M. Eichberg, M. Reif, B. Hermann, J. Lerch, and M. Mezini, "Codematch: obfuscation won't conceal your repackaged app," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 638–648.

[29] L. Jiang, J. An, H. Huang, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "Binaryai: Binary software composition analysis via intelligent binary source code matching," 2024.

[30] S. Ahn, S. Ahn, H. Koo, and Y. Paek, "Practical binary code similarity detection with bert-based transferable similarity learning," ser. ACSAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 361–374.

[31] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "jtrans: jump-aware transformer for binary code similarity detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery,

2022, p. 1–13.

[32] S. Sheng, Y. Xu, T. Zhang, Z. Shen, L. Fu, J. Ding, L. Zhou, X. Gan, X. Wang, and C. Zhou, “RepEval: Effective text evaluation with LLM representation,” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 7019–7033. [Online]. Available: https://aclanthology.org/2024.emnlp-main.398

[33] V. Cochard, D. Pfammatter, C. T. Duong, and M. Humbert, “Investigating graph embedding methods for cross-platform binary code similarity detection,” in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, 2022, pp. 60–73.

[34] A. Qasem, M. Debbabi, B. Lebel, and M. Kassouf, “Binary function clone search in the presence of code obfuscation and optimization over multi-cpu architectures,” in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 443–456.

[35] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, “An empirical comparison of pre-trained models of source code,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2136–2148.

[36] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “Graphcodebert: Pre-training code representations with data flow,” arXiv preprint, September 2020.

[37] M.-A. Lachaux, B. Roziere, M. Szafraniec, and G. Lample, “Dobf: A deobfuscation pre-training objective for programming languages,” in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 14 967–14 979.

[38] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “Unified pre-training for program understanding and generation,” in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tur, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds. Online: Association for Computational Linguistics, Jun. 2021, pp. 2655–2668.

[39] J. Zhang, H. Hong, Y. Zhang, Y. Wan, Y. Liu, and Y. Sui, “Disentangled code representation learning for multiple programming languages,” in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Online: Association for Computational Linguistics, Aug. 2021, pp. 4454–4466. [Online]. Available: https://aclanthology.org/2021.findings-acl.391

[40] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[41] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, “Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation,” 2021.

[42] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, “UniXcoder: Unified cross-modal pre-training for code representation,” in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 7212–7225. [Online]. Available: https://aclanthology.org/2022.acl-long.499

[43] S. Biderman, H. Schoelkopf, Q. Anthony, H. Bradley, K. O’Brien, E. Hallahan, M. A. Khan, S. Purohit, U. S. Prashanth, E. Raff, A. Skowron, L. Sutawika, and O. van der Wal, “Pythia: A suite for analyzing large language models across training and scaling,” 2023.