Forensic Science International: Digital Investigation 49 (2024) 301764

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS USA 2024 - Selected Papers from the 24th Annual Digital Forensics Research Conference USA

Compiler-provenance identification in obfuscated binaries using vision transformers

Wasif Khan [a], Saed Alrabaee [a,e,f,*], Mousa Al-kfairy [b], Jie Tang [c], Kim-Kwang Raymond Choo [d]

[a] Information Systems & Security, United Arab Emirates University, 15551, Al Ain, United Arab Emirates
[b] College of Technological Innovation, Zayed University, United Arab Emirates
[c] Department of Computer Science, Tsinghua University, China
[d] Department of Information Systems and Cyber Security, University of Texas at San Antonio, USA
[e] Big Data Analytics Center, United Arab Emirates University, 15551, Al Ain, United Arab Emirates
[f] Cybersecurity Research Group Center, College of IT, United Arab Emirates University, 15551, Al Ain, United Arab Emirates

ARTICLE INFO

Keywords:
Reverse engineering
Compiler provenance
Binary code analysis
Malware analysis

ABSTRACT

Extracting compiler-provenance-related information (e.g., the source of a compiler, its version, its optimization settings, and compiler-related functions) is crucial for binary-analysis tasks such as function fingerprinting, detecting code clones, and determining authorship attribution. However, the presence of obfuscation techniques has complicated the efforts to automate such extraction. In this paper, we propose an efficient and resilient approach to provenance identification in obfuscated binaries using advanced pre-trained computer-vision models. To achieve this, we transform the program binaries into images and apply a two-layer approach for compiler and optimization prediction. Extensive results from experiments performed on a large-scale dataset show that the proposed method can achieve an accuracy of over 98 % for both obfuscated and deobfuscated binaries.

1. Introduction

Program provenance refers to the detailed aspects involved in the development of a target binary; this encompasses the tools and libraries used, along with their specific versions. A particular aspect of this, known as "compiler-provenance identification," concentrates on extracting detailed information about the compiler itself, which includes its family, version, and level of optimization. This information is pivotal, for example, to understanding the origin and distinct characteristics of a malware binary.

Existing approaches use different types of features to determine the compiler provenance. Syntactic features have been used to quantify the occurrence of a program's attributes in an assembly such as idioms, N-grams, and N-perms. Conversely, semantic features are obtained through more sophisticated analyses, such as by extracting graph-based combined features and machine learning (ML)-based embedding representations. These include graphlets, control flow graphs (CFGs), compiler transformation profiles (CTPs), and compiler tags (CTs). Additionally, structural features encapsulate the control structures or data flows within a program, for example using annotated CFGs (ACFGs). A brief description of these features is shown in Table 1.

After the feature-extraction process, various ML algorithms can be employed to ascertain compiler provenance based on the derived features. Feature selection also plays a crucial role, isolating the top-*k* salient features to enhance prediction accuracy. Despite the promising efficacy of ML-based approaches in compiler-provenance identification, their reliance on specialized, handcrafted features is noteworthy. The extraction of such features necessitates domain-specific expertise and problem-specific knowledge, presenting a significant challenge. Moreover, feature-selection techniques are not without their subjective biases and inherent limitations [1, 2].

Deep-learning (DL) algorithms can facilitate the extraction of useful features with minimal preprocessing (LeCun et al., 2015; Li et al., 2021; Voulodimos et al., 2018; Zhao et al., 2019), but they rely on a significant amount of training data. Obtaining such data is, however, time-consuming and expensive. Moreover, the difficulty in generalizing DL models across different datasets—given the changing behavior of binaries—is a major limitation. Program binaries, which are represented

* Corresponding author. Information Systems & Security, United Arab Emirates University, 15551, Al Ain, United Arab Emirates.
E-mail addresses: kwasif@uaeu.ac.ae (W. Khan), salrabaee@uaeu.ac.ae (S. Alrabaee), mousa.al-kfairy@zu.ac.ae (M. Al-kfairy), jietang@tsinghua.edu.cn (J. Tang), raymond.choo@fulbrightmail.org (K.-K. Raymond Choo).

https://doi.org/10.1016/j.fsidi.2024.301764

Available online 5 July 2024

Table 1
Features used for determining compiler provenance. CTP: compiler transformation profile; CFG: control flow graph; CCT: compiler constructor terminator; CT: compiler tag; CF: compiler function; ACFG: annotated CFG.

Feature name	Feature type	Brief description
Idiom	Syntactic	Idioms are instruction sequences in assembly code signifying specific programming constructs, helping to reveal the code's high-level structure.
N-gram	Syntactic	N-grams represent sequences of *N* instructions in the binary code. They capture local patterns and dependencies between instructions.
N-perm	Syntactic	N-perms involve considering permutations of *N* instructions. They capture the ordering of instructions without enforcing strict adjacency.
Graphlet	Semantic	Graphlets are small subgraphs within the CFG or data flow graph.
CTP	Semantic	CTPs are related to the transformations applied by the compiler during the compilation process.
CFG	Semantic	CFGs represent the flow of control between basic blocks in the binary.
CCT	Semantic	CCTs show the relationships between constructor and terminator functions added by the compiler.
CT	Semantic	CTs are annotations embedded into the binary code by the compiler.
CF	Semantic	CFs refer to specific functions or routines added by the compiler during the compilation process. These functions may serve various purposes, including runtime support or implementing certain optimizations.
ACFG	Structural	ACFGs are an extension of traditional CFGs in which additional annotations, such as compiler-related details, are included.

as strings of zeros and ones, can be transformed into matrices or images (Nataraj et al., 2011). An alternative and promising approach to compiler-provenance identification thus involves converting these binaries into images and treating the identification of their compilers as a computer-vision problem.

In this paper, we focus on deep vision for the following reasons.

- **Dealing with obfuscated binaries.** Previous studies have not generally sought to work with obfuscated binaries, and approaches designed for deobfuscated binaries experience a significant accuracy drop (He et al., 2022). However, this limitation can be mitigated when dealing with images, as they rely only on textural information, as demonstrated by (Nataraj et al., 2011).
- **Size of training datasets.** DL models trained on large-scale datasets such as ImageNet (Deng et al., 2009; Krizhevsky et al., 2012) can be effectively fine-tuned for specific tasks with minimal additional training data. These pre-trained networks serve as powerful feature extractors that capture generalizable patterns. Multiple pre-trained models that have been validated on large-scale datasets demonstrate good performance across various image-recognition tasks (LeCun et al., 2015; Voulodimos et al., 2018) (Zhao et al., 2019). Additionally, the incorporation of data-augmentation techniques prevents overfitting and increases the available training data; this is particularly important when dealing with smaller datasets (Krizhevsky et al., 2012).
- **Multiple architectures.** Previous approaches to compiler-provenance extraction have been designed to handle a particular architecture or to consider each architecture separately (Rosenblum et al., 2010, 2011; Rahimian et al., 2015; Alrabaee et al., 2020; He et al., 2022; Kim et al., 2023). However, we show that since binaries from different platforms transformed into images have similar textural information, the source of a program binary compiled with any target architecture can be identified using the same model with high accuracy.
- **Efficiency.** Transformer models (Vaswani et al., 2017) have also been extended to vision transformer (ViT) architectures (Dosovitskiy et al., 2020), which use an attention mechanism to partition an image into patches and feed the resulting sequence of linear embeddings of these patches into a transformer model. The performance of this approach surpasses that of conventional convolutional neural network (CNN) models in various computer-vision tasks (Dosovitskiy et al., 2020). Herein, we use ViT models for compiler-provenance identification.

This study focused on only the GCC and Clang compilers because they are the most commonly used cross-platform compilers (He et al., 2022); other compilers are only compatible with a few target architectures (Kim et al., 2022). Specifically, we propose a novel method to predict the compiler family and optimization level using 8 pre-trained networks and state-of-the-art transformer models. The compiler family and optimization level is predicted using multiple pre-trained DL and transformer models from the largest publicly available dataset.

This is one of the first attempts to employ pre-trained and ViT-based models for identifying compiler provenance from mixed-architecture binaries. We show that the proposed approach achieves promising performance for both obfuscated and deobfuscated binaries.

2. Related approaches

Multiple studies have examined the issue of compiler provenance. One of the first works was that of (Rosenblum et al., 2010), who focused on identifying the source of the compiler. Their work was then extended to predicting toolchain provenance (e.g., compiler family, source language, and compilation options), with promising results (Rosenblum et al., 2011). The approach outlined by (Rahimian et al., 2015) showed that extracting different types of features—syntactic, semantic, and structural—improves the predictive performance of compiler-provenance identification. The authors then proposed an ML-based approach to predict compiler version and optimization level, which was found to achieve good results. Although these works represent some of the leading approaches to compiler-provenance identification, they nonetheless rely on handcrafted features.

(Pizzolotto and Inoue, 2020) evaluated two DL-based models—a CNN model and long short-term memory (LSTM) model—to predict the compiler (GCC or Clang) and its optimization settings from a dataset of 76,000 binaries. Their experimental results showed that the CNN achieved an F-score of 0.99 for binary optimization and 0.98 for the compiler used. The authors also showed that the CNN model was better than the LSTM model because the former provided better accuracy and was easy to train (Otsubo et al., 2020a). proposed the "o-glasses" approach to visualize the *x*86 native code (program-code vs non-code) using a 1D CNN model, which was found to perform well. An extension of o-glasses, o-glassesX (Otsubo et al., 2020b), uses an attention mechanism for identifying compiler provenance, and this was also found to produce promising results, with an accuracy of more than 0.98 in identifying compiler family, optimization, and architecture (Benoit et al., 2021). introduced a graph-neural-network approach for identifying toolchain provenance that was also found to have good performance (Tian et al., 2021). applied a neural-modeling-based compiler-identification approach using CNN and recurrent neural network (RNN) models with an attention mechanism. The authors used a dataset of over 854,858 functions (4810 binaries) and achieved accuracy levels of 98.6 %, 95.3 %, and 88.7 % in identifying the compiler family, optimization level, and compiler version, respectively (He et al., 2022). proposed BinProv, which uses Bidirectional Encoder Representations from Transformers (BERT)-based embedding for compiler and optimization prediction. The authors used a subset of the BinKit dataset (Kim et al., 2022) for evaluation. A summary of the studies conducted in relation to identifying compiler and optimization provenance is presented in Table 2.

Table 2
Comparative summary of the compiler and optimization provenance literature (Syn): Syntactic (Sem): Semantic (Str): Structural (Auto): Automatic.

Work	Features				Algorithm		Analysis		Compilers	Target Architecture
	Syn	Sem	Str	Auto[a]	ML	DL	Static	Dynamic		
Rosenblum et al. (2010)	✓	×	×	×	✓	×	✓	×	GCC, ICC, MSVS	Intel IA-32
Rosenblum et al. (2011)	✓	✓	×	×	✓	×	✓	×	GCC, ICC, MSVS	Intel IA-32
Rahimian et al. (2015)	✓	✓	✓	×	✓	×	✓	×	GCC, ICC, MVS, Clang	Intel *x86*/x86-64
Chaki et al. (2011)	✓	✓	×	×	✓	×	✓	×	VS	–
Otsubo et al. (2020a)	×	×	×	✓	×	✓	✓	×	GCC	*x86*
Otsubo et al. (2020b)	×	×	×	✓	×	✓	✓	×	VS, GCC, Clang, ICC	*x86*/x86-64
Benoit et al. (2021)	×	✓	×	×	×	✓	✓	×	GCC, ICC, MVS, Clang, MinGW	Ubuntu, *x64*
Pizzolotto and Inoue (2020)	×	×	×	✓	×	✓	✓	×	GCC, Clang	*x86* 64
Lin and Gao (2021)	×	✓	×	×	–	–	✓	×	Clang, GCC	*x86*-64
He et al. (2022)	×	×	×	✓	×	✓	✓	×	GCC, Clang	*x86*/64
Otsubo et al. (2022)	×	×	×	✓	×	✓	✓	×	VC, ICC, GCC, Clang	*x86*/x86-64
Pei et al. (2021)	×	×	×	✓	×	✓	✓	×	GCC, Clang	ARM, MIPS, *x86*, *x64*
Kim et al. (2023)	✓	✓	×	×	✓	×	✓	×	GCC, Clang	ARM
Du et al. (2022)	×	✓	×	×	✓	×	✓	✓	GCC, ICC, Clang	Linux

[a] Does not require feature extraction such as DL-based algorithms.

Table 3
Architecture details of the selected DL models.

Model	Size (MB)	GFLOPS[a]	Parameters	Convolution layers	Fully connected layers
AlexNet	233.1	0.71	62M	5	3
VGG16	527.8	15.47	138M	13	3
ResNet	44.7	1.81	25M	34	1
GoogleNet	49.7	1.50	6M	22	1
DenseNet	30.8	2.83	8M	4[b]	1
MobileNet	13.6	0.30	3.5M	32	1

[a] GFLOPS: giga floating-point operations per second.
[b] Dense blocks.

3. Proposed methodology

A flowchart of the proposed methodology is shown in Fig. 1. This consists of four different modules: the dataset of program binaries, image construction from the program binaries, DL models, and the results of compiler and optimization-level prediction (see Table 3).

3.1. Program binaries dataset

We used BinKit (Kim et al., 2022), a large-scale binary-code similarity analysis benchmark consisting of over 200,000 binaries compiled from 51 GNU software packages. This contains 1351 combinations of compilers, compilation options, and target architectures. The binaries are compiled for eight different architectures from nine different compiler versions, including GCC and Clang, with five optimization levels (O0 to O3, Os). Details about the dataset can be found in the report of (Kim et al., 2022).

3.2. Obfuscated binaries dataset

In addition to the normal dataset, we included the obfuscated binaries of the BinKit dataset. The obfuscation was conducted with the commonly applied Obfuscator-LLVM system (Junod et al., 2015), using its latest version with four obfuscation options: instruction substitution (SUB), bogus control flow (BCF), control flow flattening (FLA), and a combination of all options. Each obfuscation method was treated as a distinct compiler during evaluation, and obfuscation was applied only once to prevent significant increases in binary size, which might make it challenging to process them using tools such as IDA Pro.[1] For instance, applying obfuscation twice on the `a2ps` binary with all three options results in a file that is 30 times larger when compared to the original binary (Kim et al., 2022).

3.3. Problem formulation and image dataset

Traditional methods often rely on static analysis, which may not capture the inherent complexities of binary executables. To address this, our approach involves transforming binary files into visual representations, which can be analyzed using advanced DL models.

Consider a binary file F consisting of a sequence of bytes $b_1, b_2, \ldots, b_n$, where each byte b_i (for $1 \leq i \leq n$) is an integer value in the range [0, 255]. The bytes in F are grouped into triplets to form RGB values. Each triplet (b_i, b_{i+1}, b_{i+2}) is mapped to a pixel P in the RGB image, where $P = (R, G, B)$ and $R = b_i$, $G = b_{i+1}$, and $B = b_{i+2}$. This process can be mathematically represented as:

$$P_j = (b_{3j-2}, b_{3j-1}, b_{3j})$$

for $1 \leq j \leq \lceil \frac{n}{3} \rceil$.

The dimensions of the resulting image are a function of the total number of bytes in F. Let W and H represent the width and height of the image, respectively. The value of W is determined based on the file size, and H is adjusted accordingly. This is represented as:

$$W = f(\text{size}(F)), \quad H = \frac{\lceil \frac{n}{3} \rceil}{W},$$

where f is a function that determines the width based on the file size(F).

Each RGB triplet is mapped to a pixel in the image. This mapping can be represented as a function M from the set of triplets to a set of pixels in the image grid:

$$M : \{P_j\} \rightarrow \text{Image Grid}.$$

After creating the image dataset, it will be used to train and validate several state-of-the-art DL models, which are described in the next subsections.

3.4. DL models

In this section, we describe several pre-trained models that were used in this study for compiler-provenance identification. All the pre-trained DL models are based on CNNs, which provide a backbone of DL architectures. The basic structure of a CNN consists of several layers that automatically extract features from input data. These layers include convolutional layers, max-pooling layers, and fully connected layers. A brief explanation of CNNs is now presented.

[1] https://hex-rays.com/ida-pro/.

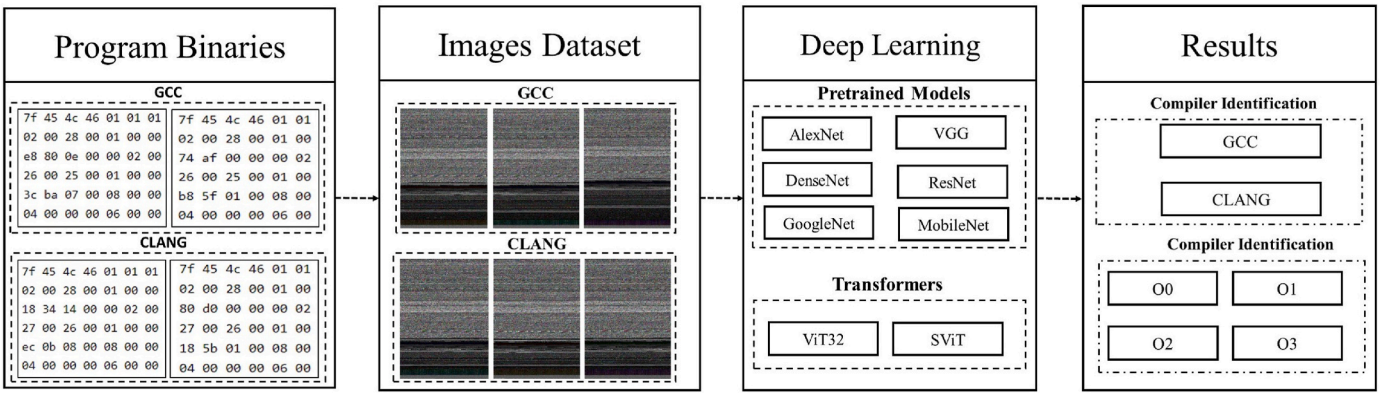


Fig. 1. Schematic flowchart of the proposed framework.

3.4.1. CNNs

CNNs are DL models that are widely used for image-related tasks, such as image classification, object detection, and segmentation. They are designed to automatically and adaptively learn hierarchical representations from raw input data.

3.4.1.1. Convolutional layers. Convolutional layers convolve image pixels with learnable filters to capture local patterns in the image data and extract useful features from the input layer:

$$y(x) = f\left(\sum_{i=1} nw_i{}^* x_i + b\right), \tag{1}$$

where x is the input, w_i represents the learnable weights, b is the bias term, * denotes the convolution operation, and f is the activation function.

3.4.1.2. Activation function. The activation function introduces non-linearity to the model and enhances the usability of the feature maps obtained from the convolution layers. The rectified linear unit (ReLU) is a commonly used activation function that aids the capture of complex relationships and increases the speed of convergence during training. It can be defined as:

$$\mathrm{ReLU}(x) = \max(0, x). \tag{2}$$

3.4.1.3. Pooling layers. A pooling layer downsamples the spatial dimensions, thus reducing computational complexity. For instance, max-pooling operates by selecting the maximum value within a local region and forwarding it to the next layer.

3.4.1.4. Fully connected layers. In a fully connected layer, the features obtained from previous layers are flattened to a 1D feature vector, which is then used for classification.

3.4.1.5. Softmax layer. A softmax layer is a final layer that is used to classify the instances based on the features obtained from the fully connected layer. For K classes, the softmax function is given by:

$$P(\mathrm{class}_i) = \frac{ez_i}{\sum_{j=1} Kez_j}, \tag{3}$$

where $P(\mathrm{class}_i)$ is the probability of the input belonging to class i, z_i is the raw output for class i, and K is the total number of classes.

3.4.1.6. Dropout regularization. Dropout is a regularization technique in which a fraction of input units are set to zero during training. This reduces the co-dependency between neurons and avoids overfitting. It can be represented as:

$$\mathrm{output} = \frac{\mathrm{input}}{1 - \mathrm{dropout\_rate}}. \tag{4}$$

Since all the pre-trained models are based on CNN architecture, we now briefly explain the DL models used in this study.

3.4.2. AlexNet

AlexNet was one of the first DL models to be trained on the large-scale ImageNet dataset, which contains more than 15 million (M) images with 1000 classes from 22,000 categories. The architecture of AlexNet contains five convolution layers, three fully connected layers, and 60M parameters.

3.4.3. VGG16

Visual geometry group 16 (VGG16) (Simonyan and Zisserman, 2014) consists of small receptive fields (3×3) with 16 layers—13 convolution layers and three fully connected layers—and it contains 138M parameters.

3.4.4. ResNet

(He et al., 2016) demonstrated the challenges associated with training deeper neural networks. Therefore, the residual network (ResNet) architecture was introduced, incorporating residual learning blocks with skip connections. These blocks enable the flow of information from one layer directly to another, skipping one or more intermediate layers. ResNet is computationally cheaper than other models such as VGG16, and it achieves better classification performance.

3.4.5. GoogleNet

For GoogleNet (Szegedy et al., 2015), incorporated an inception module into a CNN; this employs multiple parallel convolutional filters of different sizes within the same layer.

3.5. Transformers

(Vaswani et al., 2017) originally developed transformers for sequence modeling, and they show significant advances in natural language processing (NLP) tasks. Transformers address the key limitation of RNNs, which is that they process inputs sequentially. Transformers use an attention mechanism that is capable of processing sequences in parallel, making them more efficient and also faster. After the success of transformers, they were then extended to the computer-vision domain, also resulting in significant improvements. We now briefly explain the transformer used in our study.

3.5.1. ViTs

ViTs (Dosovitskiy et al., 2020) convert an image into patches (treating them as tokens) and input a sequence of linear embeddings

from these patches into a transformer. A ViT processes 2D images by reshaping them into flattened 2D patches, which are linearly projected to embeddings:

$$z_0 = [x_{\text{class}}; x_{1pE}; x_{2pE}; \ldots; x_{NpE}] + E_{\text{pos}}, \quad (5)$$

where x_{class} is the learnable embedding, x_{ipE} are the flattened patches, and E_{pos} are the position embeddings. The sequence is then input to a transformer encoder, which consists of multiheaded self-attention (MSA) and multilayer perceptron (MLP) blocks. CNN models work under the assumption that image features such as 2D neighborhood structure, locality, and translation equivariance are embedded in every layer across the model. However, in ViT, MLP blocks exhibit local and translational equivariance, while MSA blocks operate globally.

3.5.2. Swin transformer

Transformers use fixed-scale word tokens for NLP tasks. However, vision tasks vary in scale; therefore, fixed-scale tokens are not suitable for vision tasks. Furthermore, the high pixel resolution in images poses computational-complexity challenges for the self-attention mechanism of a ViT; hence, its challenging to adopt ViT in tasks requiring pixel-level dense predictions. Swin transformer (SViT) was proposed to address these challenges (Liu et al., 2021). SViT constructs hierarchical feature maps from smaller patches and then merges these patches with neighbors in the deep layers. The image is then converted into non-overlapping patches (tokens) with a feature dimension (denoted as C) of 48 ($4 \times 4 \times 3$) after a linear embedding layer. The SViT blocks are applied on the patch token.

The network employs patch merging in deeper layers to create a hierarchical representation, reducing the number of tokens through concatenation and linear-layer application. For instance, initially, it is $\left(\frac{H}{4} \times \frac{W}{4}\right)$; then it becomes $\left(\frac{H}{8} \times \frac{W}{8}\right)$, $\left(\frac{H}{16} \times \frac{W}{16}\right)$, and $\left(\frac{H}{32} \times \frac{W}{32}\right)$. In SViT, the MSA is replaced with a shifted window-based MSA module represented as:

$$\Omega(\text{W} - \text{MSA}) = 4hwC2 + 2M2hwC,$$

where M is the window size and $h \times w$ are the patches of an image. We used ViT and SViT and their variants, as shown in Table 4.

4. Experimental setup and results

This section explains the experimental results of compiler-provenance identification. We first show the compiler-identification results, and this is followed by the optimization-identification results with different combinations. The dataset was divided into training (0.60), validation (0.20), and testing sets (0.20). All the results in the tables are shown for the testing set. The experiments were conducted with an NVIDIA Tesla V100 GPU with 32 GB RAM.

4.1. Image dataset

The conversion of images from program binaries is discussed in Section 3.1. A sample of each from each compiler and their respective optimizations is shown in Fig. 2.

4.2. Compiler identification

The experimental results shown in Table 5 show that ViT achieved the best classification performance, with accuracy, precision, recall, and F-score values all equal to 0.993; this was followed by VGG16, which had respective values of 0.987, 0.985, 0.986, and 0.985. It can be seen that other models also achieved comparable performance for compiler identification.

Table 4
Model specifications for ViT models.

Model	Size (MB)	GFLOPS[a]	Parameters (M)	Patch size
ViT	330	15.38	86	224 × 224
SViT	108.2	4.49	28	224 × 224

[a] GFLOPS: giga floating-point operations per second.

4.3. Optimizations

Once the origin compiler of the program binary is identified, we can then predict the optimization used for that binary. The results for predicting the optimization associated with each binary are now presented. We first show the optimization levels for Clang and then for GCC.

4.3.1. Clang

4.3.1.1. Low and high optimization (O0 and O3). The experimental results for classifying between optimization levels O0 and O3 are presented in Table 6. This shows that VGG16 achieved the best classification performance, with accuracy, precision, recall, and F-score values of 0.945, 0.942, 0.948, and 0.945, respectively. ResNet, GoogleNet, DenseNet, and MobileNet also achieved similar performance. It can be seen that transformer models achieved inferior performance when compared to the pre-trained models, with the best performance achieved by the SViT model, which had an accuracy of 0.902.

4.3.1.2. Low and high optimization (O3 and Os). The experimental results for classifying between optimization levels O3 and Os are presented in Table 7. This shows that AlexNet, ResNet, and VGG16 exhibited relatively higher accuracy values: 0.607, 0.645, and 0.671, respectively. VGG16 showed higher precision, recall, and F-score values: 0.671, 0.711, and 0.685, respectively. The recall of DenseNet was the best at 0.840; however, the precision was only 0.629. The transformer models performed the worst in comparison to the pre-trained models.

4.3.1.3. Low and high optimization (O0 and Os). The experimental results in Table 8 show that none of the classifiers was able to achieve better classification performance in this case. The Swin transformer achieved the best performance, and this was slightly better than the random classifier.

4.3.1.4. Multi-level optimization (O0, O1, and O2). Table 9 presents the results of a multi-class classification approach to distinguishing the O0, O1, and O2 optimization levels. This shows that VGG16 achieved the best classification performance, with accuracy, precision, recall, and F-score values of 0.811, 0.810, 0.811, and 0.811, respectively. Other pre-trained models also achieved comparable performance; however, transformer-based models remained the worst.

4.3.1.5. Multi-level optimization for all cases (O0, O1, O2, O3, and Os). Table 10 presents the prediction results across all the optimization levels for the Clang compiler. This shows that ViT achieved the highest performance, with accuracy values of 0.659, 0.656, 0.662, and 0.659, respectively.

4.3.2. GCC

The experimental results to predict different optimization levels for the GCC compiler are now presented.

4.3.2.1. Low and high optimization (O0 and O3). Table 11 shows the results of classification between the O0 and O3 optimization levels for the GCC compiler. It can be seen that VGG16 consistently achieved the best classification performance when compared to the other models, with an accuracy of 0.918. For the transformer-based model, SViT achieved better performance, with an accuracy of 0.869.

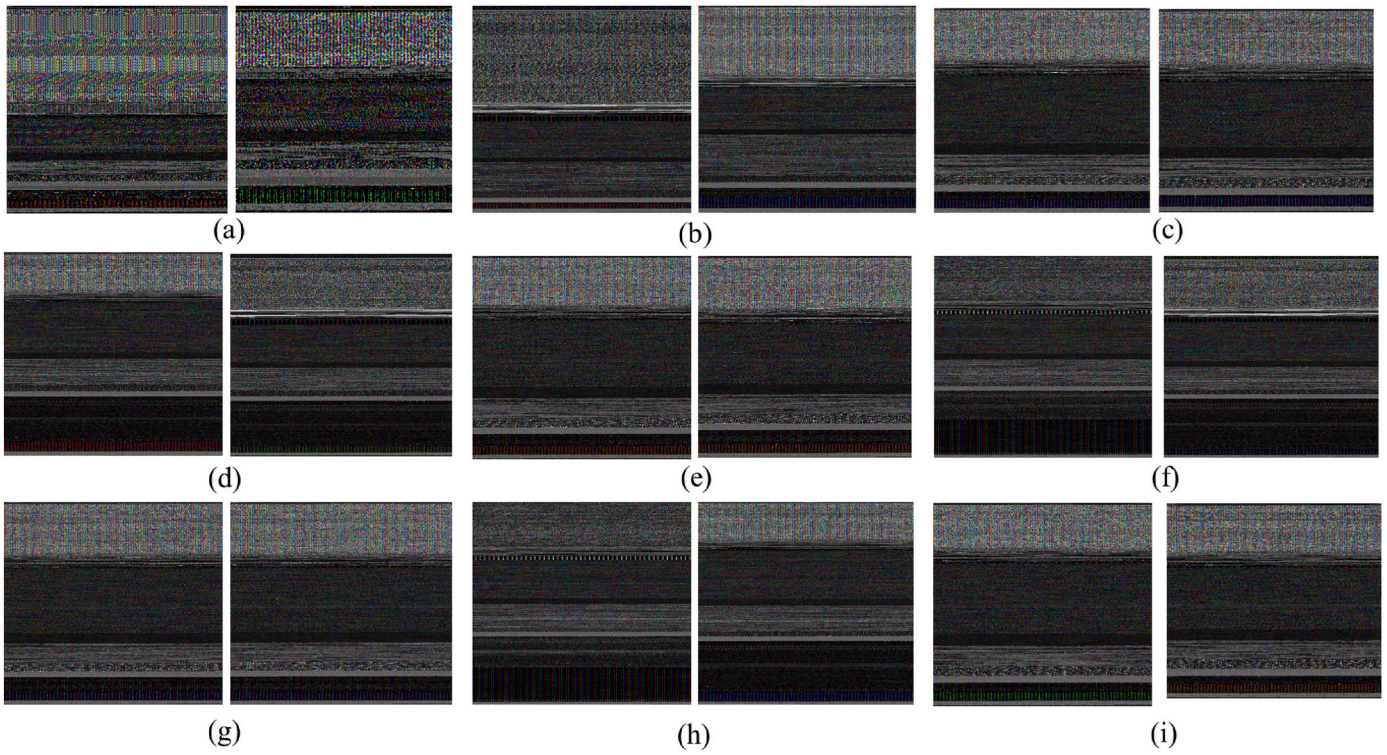


Fig. 2. Samples of images obtained from binaries: (a) GCC O0; (b) Clang O0; (c) GCC O1; (d) Clang O1; (e) GCC O2; (f) Clang O2; (g) GCC O3; (h) Clang O3; (i) GCC Os.

Table 5
GCC and Clang identification using DL models.

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.961	0.957	0.957	0.957
ResNet	0.978	0.975	0.974	0.975
VGG16	0.987	0.985	0.986	0.985
GoogleNet	0.980	0.979	0.975	0.977
ViT	0.993	0.993	0.993	0.993
Swin T	0.969	0.970	0.969	0.969

Table 6
Clang optimization O0 and O3.

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.896	0.900	0.895	0.898
ResNet	0.930	0.928	0.930	0.929
VGG16	0.945	0.942	0.948	0.945
GoogleNet	0.913	0.947	0.875	0.909
DenseNet	0.925	0.937	0.915	0.926
MobileNet	0.920	0.927	0.909	0.918
ViT	0.814	0.823	0.790	0.806
SViT	0.902	0.902	0.902	0.902

Table 7
Clang optimization O3 and Os.

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.607	0.602	0.675	0.636
ResNet	0.645	0.654	0.620	0.637
VGG16	0.671	0.661	0.711	0.685
GoogleNet	0.631	0.608	0.730	0.664
DenseNet	0.631	0.629	0.629	0.629
MobileNet	0.600	0.564	0.840	0.675
ViT	0.492	0.454	0.061	0.108
SViT	0.522	0.545	0.522	0.533

Table 8
Clang optimization O0 and Os.

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.491	0.494	0.322	0.390
ResNet	0.495	0.499	0.507	0.503
VGG16	0.491	0.500	0.434	0.464
GoogleNet	0.498	0.483	0.434	0.457
DenseNet	0.500	0.507	0.167	0.252
MobileNet	0.495	0.492	0.817	0.614
ViT	0.500	0.50	0.50	0.50
SViT	0.506	0.517	0.506	0.511

Table 9
Clang optimization O0, O1, and O2.

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.725	0.723	0.725	0.724
ResNet	0.779	0.777	0.779	0.778
VGG16	0.811	0.810	0.811	0.811
GoogleNet	0.753	0.754	0.753	0.754
DenseNet	0.768	0.774	0.768	0.771
MobileNet	0.746	0.758	0.746	0.752
ViT	0.538	0.525	0.538	0.532
SViT	0.674	0.671	0.674	0.672

4.3.2.2. Low and high optimization (O3 and Os). The experimental results for classification between O3 and Os are presented in Table 12. These show that ResNet achieved higher performance across all metrics, with an accuracy of 0.869, precision of 0.871, recall of 0.869, and F-score of 0.870. VGG16 and DenseNet also achieved comparable performance; however, the transformer-based models performed the worst.

4.3.2.3. Multi-level optimization (O0 and Os). For O0 and Os optimization prediction, VGG16 performed the best, as shown in Table 13.

Table 10
Clang optimization for all levels.

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.483	0.469	0.483	0.476
ResNet	0.533	0.515	0.533	0.524
VGG16	0.575	0.566	0.575	0.571
GoogleNet	0.516	0.500	0.516	0.508
DenseNet	0.508	0.530	0.508	0.519
MobileNet	0.495	0.545	0.495	0.519
ViT	0.659	0.656	0.662	0.659
SViT	0.479	0.460	0.479	0.469

Table 11
Optimizations for GCC: O0 and O3.

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.866	0.866	0.866	0.866
ResNet	0.905	0.905	0.905	0.905
VGG16	0.918	0.918	0.918	0.918
GoogleNet	0.887	0.889	0.887	0.888
DenseNet	0.896	0.897	0.896	0.897
MobileNet	0.899	0.900	0.899	0.900
ViT	0.848	0.848	0.848	0.848
SViT	0.922	0.922	0.922	0.922

Table 12
Optimizations for GCC: O3 and Os.

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.718	0.719	0.718	0.718
ResNet	0.869	0.871	0.869	0.870
VGG16	0.780	0.781	0.780	0.780
GoogleNet	0.736	0.754	0.736	0.745
DenseNet	0.758	0.771	0.758	0.764
MobileNet	0.727	0.744	0.727	0.735
ViT	0.727	0.728	0.727	0.728
SViT	0.661	0.670	0.661	0.665

Table 13
Optimizations for GCC: O0 and Os.

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.852	0.854	0.852	0.853
ResNet	0.869	0.871	0.869	0.870
VGG16	0.911	0.911	0.911	0.911
GoogleNet	0.876	0.876	0.876	0.876
DenseNet	0.861	0.864	0.861	0.863
MobileNet	0.869	0.876	0.869	0.873
ViT	0.957	0.957	0.957	0.957
SViT	0.887	0.889	0.887	0.887

Table 14
Optimizations for GCC: O0, O1, and O2.

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.667	0.671	0.667	0.669
ResNet	0.698	0.698	0.698	0.698
VGG16	0.742	0.741	0.742	0.742
GoogleNet	0.697	0.696	0.697	0.697
DenseNet	0.712	0.714	0.712	0.713
MobileNet	0.697	0.708	0.697	0.702
ViT	0.852	0.852	0.852	0.852
SViT	0.718	0.718	0.718	0.718

Table 15
GCC: optimizations at all levels.

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.493	0.490	0.493	0.491
ResNet	0.507	0.501	0.507	0.504
VGG16	0.547	0.545	0.547	0.546
GoogleNet	0.420	0.406	0.420	0.413
DenseNet	0.518	0.516	0.518	0.517
MobileNet	0.514	0.510	0.514	0.512
ViT	0.717	0.718	0.717	0.718
SViT	0.397	0.374	0.397	0.385

4.3.2.4. Multi-level optimization (O0, O1, and O2). The performance for multi-level optimization is presented in Table 14. This shows a drop in the accuracy, with the maximum accuracy of 0.741 achieved by VGG16.

4.3.2.5. Multi-level optimization for all cases (O0, O1, O2, O3, and Os). The performance for all optimization levels is represented in Table 15.

4.3.3. Obfuscated binaries

We conducted a comprehensive evaluation of our proposed model using obfuscated binaries sourced from the BinKit dataset. The BinKit dataset provides the obfuscated binaries for Clang only; therefore, we performed various experiments to identify multiple optimization levels ranging from O0 to Os.

4.3.3.1. Low and high optimization (O0 and O3). The experimental results evaluating the performance of multiple DL models for identification between O0 and O3 in obfuscated binaries are shown in Table 16. These show that the proposed technique achieved promising performance even for obfuscated binaries. The best result was achieved by ViT, which had accuracy, precision, recall, and F-score values all equal to 0.971.

4.3.3.2. Low and high optimization (O0 and Os). The experimental results for predicting O0 bs Os are presented in Table 17. This shows that the best performance was achieved by ViT, with an accuracy of 0.971.

4.3.3.3. Low and high optimization (O3 and Os). In differentiating between O3 and Os (Table 18), the best performance was achieved by VGG16, with accuracy, precision, recall, and F-score values of 0.638, 0.641, 0.638, and 0.639, respectively. Comparable performance was achieved by ViT.

4.3.4. Model convergence and t-SNE visualization for obfuscated binaries

This section presents an example of the proposed approach, in which we show the training and validation accuracies as well as losses for multiple models. Additionally, we incorporate t-distributed stochastic neighbor embedding (t-SNE) visualizations to examine the separation of data points. The training and validation accuracies shown in Fig. 3(a) reveal that nearly all the models converge after the eighth epoch, indicating their proficiency in learning patterns and distinguishing between various optimizations. Similarly, the training and validation losses exhibit consistent patterns.

Example t-SNE plots for optimization prediction (O0 and O3) with obfuscated binaries are presented in Fig. 4. It can be seen that ViT and the Swin transformer (Fig. 4(g) and (h)) show better performance, accurately identifying and segregating the binaries. While the separation of data points achieved by VGG16 and ResNet is also better, the ViT and SViT models exhibit superior performance, as can be seen from the cluster separation.

5. Discussion

This work highlights the effectiveness of vision models for compiler-provenance identification. The transformation of binaries into images shows promising results for both the obfuscated and deobfuscated cases. It can be seen that each compiler and their respective optimization levels have distinctive textures; therefore, the DL models obtained encouraging results. We showed that our approach achieved an accuracy of 99

Table 16
Performance on obfuscated binaries (O650 and 03).

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.848	0.848	0.848	0.848
ResNet	0.893	0.894	0.893	0.893
VGG16	0.928	0.928	0.928	0.928
GoogleNet	0.886	0.886	0.886	0.886
DenseNet	0.745	0.745	0.745	0.745
MobileNet	0.736	0.736	0.736	0.736
ViT	0.971	0.971	0.971	0.971
SViT	0.959	0.959	0.959	0.959

Table 17
Performance on obfuscated binaries (O0 and Os).

Model	Accuracy	Precision	Recall	F-score
AlexNet	0.850	0.850	0.850	0.850
ResNet	0.878	0.879	0.878	0.878
VGG16	0.927	0.928	0.927	0.928
GoogleNet	0.868	0.869	0.868	0.869
DenseNet	0.750	0.752	0.750	0.751
MobileNet	0.742	0.742	0.742	0.742
ViT	0.971	0.971	0.971	0.971
SViT	0.963	0.963	0.963	0.963

Table 18
Performance on obfuscated binaries (O3 and Os).

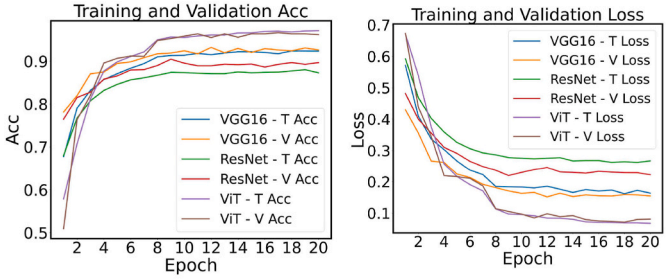
Model	Accuracy	Precision	Recall	F-score
AlexNet	0.584	0.586	0.584	0.585
ResNet	0.599	0.601	0.599	0.600
VGG16	0.638	0.641	0.638	0.639
GoogleNet	0.588	0.588	0.588	0.588
DenseNet	0.515	0.516	0.515	0.516
MobileNet	0.530	0.529	0.530	0.529
ViT	0.633	0.633	0.633	0.633
SViT	0.620	0.621	0.620	0.620

% for compiler identification. It can be seen from Table 5 that ViT achieved the best performance, with an accuracy of 0.993; this was followed by VGG16, with an accuracy of 0.987. The models' results are comparable, showing that any pre-trained models can be adopted for compiler identification.

Several studies conducted considering compiler-provenance identification have demonstrated similar performance (Rosenblum et al., 2010, 2011; Chaki et al., 2011; Otsubo et al., 2020a; He et al., 2022). However, our approach predicts the compiler in an architecture-agnostic way, i.e., regardless of the source architecture. For instance (He et al., 2022), evaluated their method only on the *x*64-86 target architecture. Furthermore, we used the latest version of the BinKit dataset (version 2.0),[2] which is the largest publicly available dataset. In contrast, some works only used a subset of the BinKit dataset for evaluation (He et al., 2022).

We also performed extensive experiments to identify the optimization level used during the compilation process. Our approach showed high performance in predicting between low (O0) and high (O3) optimization levels, with over 95 % accuracy for both GCC and Clang. We also experimented with distinguishing between various optimization levels, such as O0 vs Os, O3 vs Os, and multi-level classification (O0, O1, and O2, and O0, O1, O2, O3, and Os) for both GCC and Clang. Our analysis showed that distinguishing between low (O0) and high (O3) had the best performance; however, other optimization levels also achieved comparable performance. For instance, for GCC optimizations, the accuracy for O0 vs O3 was encouraging, with the highest accuracy

[2] https://github.com/SoftSec-KAIST/BinKit.



Training and validation accuracies Training and validation loss

Fig. 3. Sample of training and validation accuracies and loss. T Acc: Training Accuracy, V Acc: Validation accuracy, T Loss: Training Loss, V Loss: Validation loss.

achieved by ViT, at 0.962. Furthermore, the performance for O0 vs Os was also good, with an accuracy of 0.957. However, for O3 vs Os, the performance was slightly reduced to 0.869. For multi-level optimization, for distinguishing between O0, O1, and O2, the accuracy was 0.742; for distinguishing between all optimization levels, the accuracy was only 0.71, and this was achieved by ViT.

The experimental results for obfuscated binaries demonstrated the efficacy of the proposed approach, which has an accuracy as high as 0.971 using ViT for distinguishing between low (O0) and high (O3) optimization levels (see Table 16). In contrast to various existing techniques, which often struggle with obfuscated binaries, our approach exhibits resilience, achieving performance similar to deobfuscated binaries. This shows the robustness and effectiveness of vision-based models for compiler-provenance identification in both obfuscated and deobfuscated binaries.

We have also presented an illustration of model convergence and t-SNE visualizations to analyze the behavior of the DL models. The results show that all models exhibit convergence, demonstrating their potential to be effectively trained even up to 20 epochs. Moreover, the t-SNE visualizations highlight the ability to cluster data points, particularly in transformer models (Fig. 4). This shows the ability of the DL models to reveal complex patterns within the image-based program binaries.

The experimental results show that the proposed approach is architecture agnostic and does not rely on handcrafted features. Furthermore, it uses pre-trained networks without the need for extensive fine-tuning. Therefore, this approach can be adopted with minimal fine-tuning.

6. Conclusions and future work

Herein, we have proposed a novel approach to predicting the compiler family and optimization level by transforming a binary into an images and using state-of-the-art DL models. We have shown that our approach achieved over 98 % accuracy for compiler identification and over 95 % accuracy for optimization-level identification. The proposed method is simple yet efficient, and it can be used for compiler-provenance identification in an architecture-agnostic manner.

Although our approach demonstrated promising performance, we now also highlight the potential for future work. All the algorithms were used with their default parameters without any hyperparameter tuning. The performance could be further improved with extensive hyperparameter optimization. In the future, we also aim to include more compilers in addition to GCC and Clang. Furthermore, we aim to use external validation, i.e., training our model on BinKit and testing it on multiple datasets available for compiler-provenance identification to evaluate the generalizability of the proposed approach.

Data availability

We used publicly available datasets and publicly available Pytorch

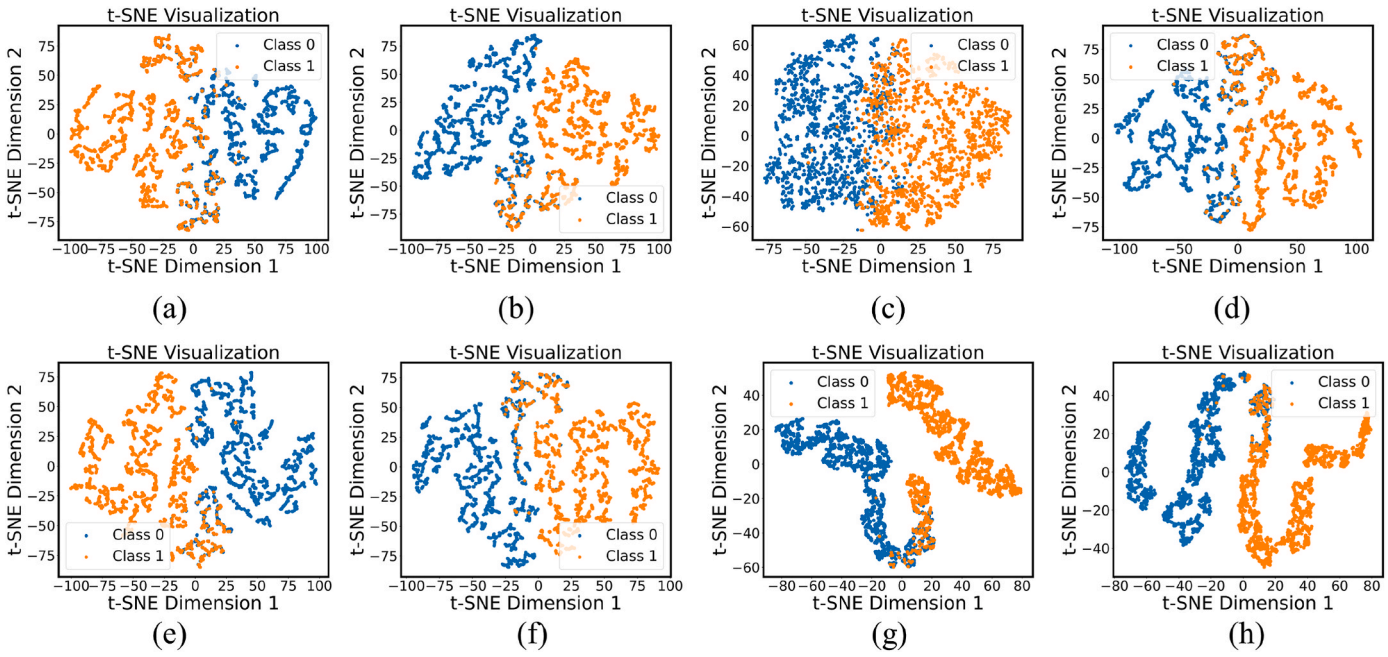


Fig. 4. t-SNE visualization in obfuscated binaries. (a): AlexNet, (b): ResNet, (c): VGG16, (d): GoogleNet, (e): DenseNet, (f): MobileNet, (g): ViT, (h): SViT

models.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We are grateful to the anonymous reviewers for their comments and suggestions. This work is supported by AUA-UAEU Joint Research Grant number 12R170.

References

Alrabaee, S., Debbabi, M., Shirani, P., Wang, L., Youssef, A., Rahimian, A., Nouh, L., Mouheb, D., Huang, H., Hanna, A., 2020. Compiler provenance attribution. In: Binary Code Fingerprinting for Cybersecurity. Springer, pp. 45–78.

Benoit, T., Marion, J.-Y., Bardin, S., 2021. Binary level toolchain provenance identification with graph neural networks. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 131–141.

Chaki, S., Cohen, C., Gurfinkel, A., 2011. Supervised learning for provenance-similarity of binaries. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 15–23.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., Fei-Fei, L., 2009. Imagenet: a large-scale hierarchical image database. In: 2009 IEEE Conference on Computer Vision and Pattern Recognition. Ieee, pp. 248–255.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al., 2020. An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale arXiv preprint arXiv: 2010.11929.

Du, Y., Snow, K., Monrose, F., et al., 2022. Automatic recovery of fine-grained compiler artifacts at the binary level. In: 2022 USENIX Annual Technical Conference (USENIX ATC 22), pp. 853–868.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778.

He, X., Wang, S., Xing, Y., Feng, P., Wang, H., Li, Q., Chen, S., Sun, K., 2022. Binprov: binary code provenance identification without disassembly. In: Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses, pp. 350–363.

Junod, P., Rinaldini, J., Wehrli, J., Michielin, J., 2015. Obfuscator-llvm–software protection for the masses. In: 2015 Ieee/acm 1st International Workshop on Software Protection. IEEE, pp. 3–9.

Kim, D., Kim, E., Cha, S.K., Son, S., Kim, Y., 2022. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. IEEE Trans. Software Eng. 49 (4), 1661–1682.

Kim, J., Genkin, D., Leach, K., 2023. Revisiting Lightweight Compiler Provenance Recovery on Arm Binaries arXiv preprint arXiv:2305.03934.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst. 25.

LeCun, Y., Bengio, Y., Hinton, G., 2015. Deep learning. Nature 521 (7553), 436–444.

Li, Zewen, Liu, Fan, Yang, Wenjie, Peng, Shouheng, Zhou, Jun, 2021. A survey of convolutional neural networks: analysis, applications, and prospects. IEEE Transact. Neural Networks Learn. Syst. 33 (12), 6999–7019 {IEEE}.

Lin, Y., Gao, D., 2021. When function signature recovery meets compiler optimization. In: 2021 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 36–52.

Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., Guo, B., 2021. Swin transformer: hierarchical vision transformer using shifted windows. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 10012–10022.

Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S., 2011. Malware images: visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security, pp. 1–7.

Otsubo, Y., Otsuka, A., Mimura, M., Sakaki, T., 2020a. o-glasses: Visualizing x86 code from binary using a 1d-cnn. IEEE Access 8, 31753–31763.

Otsubo, Y., Otsuka, A., Mimura, M., Sakaki, T., Ukegawa, H., o-glassesx, 2020b. Compiler provenance recovery with attention mechanism from a short code fragment. In: Proceedings of the 3nd Workshop on Binary Analysis Research.

Otsubo, Y., Otsuka, A., Mimura, M., 2022. Compiler Provenance Recovery for Multi-Cpu Architectures Using a Centrifuge Mechanism arXiv preprint arXiv:2211.13110.

Pei, K., Guan, J., Broughton, M., Chen, Z., Yao, S., Williams-King, D., Ummadisetty, V., Yang, J., Ray, B., Jana, S., 2021. Stateformer: fine-grained type recovery from binaries using generative state modeling. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 690–702.

Pizzolotto, D., Inoue, K., 2020. Identifying compiler and optimization options from binary code using deep learning approaches. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 232–242.

Rahimian, A., Shirani, P., Alrbaee, S., Wang, L., Debbabi, M., 2015. Bincomp: a stratified approach to compiler provenance attribution. Digit. Invest. 14, S146–S155.

Rosenblum, N.E., Miller, B.P., Zhu, X., 2010. Extracting compiler provenance from program binaries. In: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 21–28.

Rosenblum, N., Miller, B.P., Zhu, X., 2011. Recovering the toolchain provenance of binary code. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 100–110.

Simonyan, K., Zisserman, A., 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition arXiv preprint arXiv:1409.1556.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A., 2015. Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–9.

Tian, Z., Huang, Y., Xie, B., Chen, Y., Chen, L., Wu, D., 2021. Fine-grained compiler identification with sequence-oriented neural modeling. IEEE Access 9, 49160–49175.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. Adv. Neural Inf. Process. Syst. 30.
Voulodimos, A., Doulamis, N., Doulamis, A., Protopapadakis, E., et al., 2018. Deep learning for computer vision: a brief review. Comput. Intell. Neurosci. 2018.
Zhao, Z.-Q., Zheng, P., Xu, S.-t., Wu, X., 2019. Object detection with deep learning: a review. IEEE Transact. Neural Networks Learn. Syst. 30 (11), 3212–3232.