electronics

MDPI

Article

Function-Level Compilation Provenance Identification with Multi-Faceted Neural Feature Distillation and Fusion

Yang Gao [1,*], Lunjin Liang [1], Yifei Li [1], Rui Li [2] and Yu Wang [2]

[1] Department of Computer Science and Technology, Heilongjiang University, Harbin 150080, China; 20211317@s.hlju.edu.cn (L.L.); 20196762@s.hlju.edu.cn (Y.L.)
[2] School of Continuing Education, Xi'an Jiaotong University, Xi'an 710049, China; lrvberg@xjtu.edu.cn (R.L.); wy1116@xjtu.edu.cn (Y.W.)
* Correspondence: gymail2009@163.com

Abstract: In the landscape of software development, the selection of compilation tools and settings plays a pivotal role in the creation of executable binaries. This diversity, while beneficial, introduces significant challenges for reverse engineers and security analysts in deciphering the compilation provenance of binary code. To this end, we present MulCPI, short for **Mul**ti-representation Fusion-based **C**ompilation **P**rovenance **I**dentification, which integrates the features collected from multiple distinct intermediate representations of the binary code for better discernment of the fine-grained function-level compilation details. In particular, we devise a novel graph-oriented neural encoder improved upon the gated graph neural network by subtly introducing an attention mechanism into the neighborhood nodes' information aggregation computation, in order to better distill the more informative features from the attributed control flow graph. By further integrating the features collected from the normalized assembly sequence with an advanced Transformer encoder, MulCPI is capable of capturing a more comprehensive set of features manifesting the multi-faceted lexical, syntactic, and structural insights of the binary code. Extensive evaluation on a public dataset comprising 854,858 unique functions demonstrates that MulCPI exceeds the performance of current leading methods in identifying the compiler family, optimization level, compiler version, and the combination of compilation settings. It achieves average accuracy rates of 99.3%, 96.4%, 90.7%, and 85.3% on these tasks, respectively. Additionally, an ablation study highlights the significance of MulCPI's core designs, validating the efficiency of the proposed attention-enhanced gated graph neural network encoder and the advantages of incorporating multiple code representations.

Keywords: compilation provenance; graph neural network; compiler identification; feature fusion

Citation: Gao, Y.; Liang, L.; Li, Y.; Li, R.; Wang, Y. Function-Level Compilation Provenance Identification with Multi-Faceted Neural Feature Distillation and Fusion. *Electronics* **2024**, *13*, 1692. https://doi.org/10.3390/electronics13091692

Academic Editors: Kim Jindae and Seonah Lee

Received: 1 April 2024
Revised: 22 April 2024
Accepted: 22 April 2024
Published: 27 April 2024

1. Introduction

In the process of software development, a variety of toolchains and configuration settings could be utilized to convert the source code into the final executable binary. For instance, developers might select different compilers, such as GCC or LLVM, along with various compiler optimization levels such as O0 and O2, based on factors like stability, performance considerations, the size constraints of the resulting binary, and their expertise with these tools. This offers the developers great flexibility to manufacture binaries that meet customized requirements. Nevertheless, this variety in compilation toolchains also poses significant challenges to reverse engineers and security analysts, since the binary output can vary greatly across different compilation settings [1,2].

Compilation provenance identification [3–7] is a task that reversely reveals from the binary code the compilation details, like the specific compiler family, the optimization option, and even the compiler version used during the compilation phase, and has thus garnered considerable interest from researchers since it unveils critical insights into the nuances of the binary production process. Precisely pinpointing these compilation details is also crucial and advantageous for enhancing the efficacy of a myriad of binary code

analysis applications [8], such as binary code similarity detection [9,10], software plagiarism detection [11], binary safety verification [12], and program authorship attribution [13].

Of existing compilation provenance identification solutions, the signature-based methods and the learning-based approaches are two main categories that have been investigated. The signature-based approaches, as implemented in reverse engineering tools like IDA, rely on matching the entire binary against a database of expert-crafted compiler-specific signatures. These methods are hampered by the requirement of significant expertise and laborious efforts in the development of sufficiently precise compiler-specific signatures and are limited by their coarse granularity of identification. Thus, the more recent works [4,12,14,15] all follow the learning-based paradigm, by training from well-labeled samples a fine-grained classification model, later with which to infer compilation details on unseen binaries.

To ensure the robustness of the learnt models, the traditional machine learning-based ones generally enforce artificially intervened feature engineering to extract compilation-indicative features, such as idioms [14] (assembly instruction combinations incorporating wildcards) or graphlets [3] (small subgraphs derived from the CFG), from the binary code. To alleviate the human bias impacts in the feature selection process, the deep-learning (DL) based approaches [5,16,17], on the other hand, directly leverage deep neural encoders to grasp compilation-indicative features by witnessing on massive samples, and exhibit superior identification capability [4,5].

Prevailing DL-based compilation provenance identification methods generally resort to sequence-oriented neural encoders, such as CNNs [4,16] and RNNs [4,16,17], to grasp indicative features either from the raw bytes [16] or from the normalized assemblies [4,17]. These sequence-form intermediate representations of the binary code majorly imply features that reflect the impacts of diversified compilations on the lexical and syntactic aspects of the code. However, the structure of the binary code also varies significantly across different compilation settings [1]. Thus, taking into consideration the graph-form intermediate representations, such as the CFG, is likely to offer additional structural-related indicative features that are hard to be captured from sequences.

In this regard, we propose to further incorporate the informative structural features implied within graph-form intermediate representations into the final classification-inducing feature set, apart from the typically considered lexical and syntactic related features. To achieve that, we firstly employ well-adapted sequence-oriented and graph-oriented neural encoders to gather compilation-relevant features from the normalized assembly sequence and the ACFG of the binary function to be checked, respectively. Subsequently, these features are deeply integrated through a fusion strategy derived from the prevalent self-attention mechanism. This methodology enables a comprehensive capture of lexical, syntactic, and structural features that well reflect specific compilation paradigms. The primary contributions of this work are summarized as follows:

- A novel DL-based compilation provenance identification approach called MulCPI is presented, which comprehensively integrates the compilation-indicative features gathered from the binary function's normalized assembly sequence and the ACFG constructed utilizing sent2vec for enhanced capability in discerning fine-grained function-level compilation details.
- To effectively distill informative features from the ACFG, we present an advanced graph-oriented neural encoder that has improved upon the gated graph neural network (GGNN) by subtly introducing an attention mechanism to better supervise the aggregation computation of the information from the neighborhood nodes. Additionally, a fusion strategy predicated on self-attention is employed to more effectively integrate the features gathered from the ACFG with those collected from the normalized assembly sequence through the utilization of a Transformer encoder.
- Extensive experiments are conducted on a public dataset comprising 854,858 unique functions for the systematic performance evaluation of MulCPI on various tasks including identification of the compiler family, its specific version, the optimiza-

tion level, and their combination. The findings demonstrate that MulCPI exhibits impressive capability in disclosing the intricate compilation details, surpassing state-of-the-art function-level compilation provenance identification approaches in both detection accuracy and F1-scores. Moreover, the conducted ablation study confirms the value of fusing multiple distinct yet complementary code intermediate representations. The source code implementation of MulCPI has also been made public at https://github.com/gyjuice/MulCPI (accessed on 21 April 2024).

The remainder of this paper is organized as follows. Section 2 offers an overview of related work in the field of compilation provenance identification. Section 3 provides a comprehensive insight into the innovative designs of MulCPI, detailing the preparation of the two distinct intermediate code representations, their respective feature encoding methods, and the fusion strategy employed. Section 4 is dedicated to an extensive evaluation, including the experimental setup, detailed analysis of the results, and comparative observations between MulCPI and other methodologies. Section 5 acknowledges the limitations of our approach and suggests intriguing avenues for future research. Finally, the paper concludes with Section 6.

2. Related Work

Broadly speaking, existing research in compilation provenance identification fall into three main categories: those that rely on signature matching and those that employ machine or deep learning techniques.

2.1. Signature Matching-Based Approaches

The signature-based methods in compilation identification entail matching a binary program against a set of specific and carefully crafted signatures to attribute the compiler's label to the entire program based on the signature that matches. This approach has been implemented in several reverse engineering platforms, such as IDA Pro, LANGUAGE 2000 (https://farrokhi.net/language/, accessed on 21 April 2024), and PEiD (https://www.aldeid.com/wiki/PEiD, accessed on 21 April 2024). Despite being efficient, they show insurmountable limitations, including the need for specialized expertise and laborious efforts to create accurate compiler-specific signatures and the potential for reduced accuracy due to minor signature variations. Additionally, the signatures often rely on metadata or program header details, which can be easily tampered with by malicious attackers or just be missing in the stripped binaries. Moreover, these tools identify merely the compiler family from the binary, rather than much more comprehensive information [4] such as the detailed optimization level and compiler version, as MulCPI does. Furthermore, the identification outcomes generally target the entire binary, despite the possibility that a program could be compiled with multiple compilers, such as when the library code (which is compiled with a certain compilation setting) is statically linked to the main program (which adopts a different compilation setting) to produce the final binary.

2.2. Learning-Based Approaches

The learning-based approaches [3,4,18–20] treat the task as a machine learning challenge, with the assumption that the unique characteristics reflected within the binaries can reveal the specific compilation settings used for their creation by identifying the compilation-specific patterns implied within the binary code. Identification models are trained with labeled samples to recognize these patterns, which are then used to make predictions on new unseen binaries.

Rosenblum et al.'s groundbreaking study [14] introduced this methodology by defining a series of instruction idioms with placeholders to identify compiler-specific patterns through mutual information analysis, successfully determining compiler families with high accuracy. However, the method's efficacy in identifying optimization levels remains unknown due to a lack of evaluation. The subsequent ORIGIN [3] work improved upon this by incorporating graphlets to capture structural features of the code, enhancing ac-

curacy in compilation detail recovery. Hidden Markov models have also been employed to distinguish between compilers based on instruction type and frequency, necessitating distinct models for each compiler family without addressing optimization level identification [21,22]. BinComp [15] introduced a stratified technique for more resource-efficient compiler identification, but its focus on compiler-related functions limits its applicability to the broader challenge of function-level compilation provenance identification. Generally, the accuracy of these conventional machine learning methods heavily depends on the feature extraction and selection strategies carefully chosen based on expert experience, thus often introducing human bias and potentially missing relevant features.

With the deep learning techniques having demonstrated remarkable performance in addressing a variety of program analysis challenges [23–25], several studies [4,26] have also been proposed that leverage deep neural networks to identify compilation details. BinEye [16] is such an initial work, which employs instruction embeddings and CNNs to recognize the optimization levels for each object file. To pinpoint the compilation details on the finer-grained function level, Pizzolotto et al. [26] utilizes either a layered CNN or the LSTM to grasp the interesting features from the raw bytes. o-glassesX [19] also leverages a deep neural encoder comprised of a stack of CNN layers but further integrates it with the dot-product attention to enhance the feature extraction capability. NeuralCI [4], which was known as SOTA for a time, explores integrating multiple attention mechanisms into two representative neural network structures, TextCNN and BiGRU, to distill the important features from the instruction sequences, and conducts the most comprehensive and systematic evaluation. In addition to these approaches that focus on raw bytes or instruction sequences, Structure2Vec [27] explores utilizing a graph embedding network to convert the CFG of a function into vectors for training a compiler family identification classifier. Similarly, SNNN [28] operates on CFGs by simplifying them to include only the types of control flow instructions and employing the GCN for feature extraction. Unlike these methods, which typically rely on a single code intermediate representation for feature extraction, MulCPI utilizes multiple distinct intermediate representations that capture a wide range of lexical, syntactic, and structural aspects of code. It also employs sophisticated neural encoders to extract the deep hidden yet significant indicative features more comprehensively, thereby enhancing its identification capabilities.

3. Problem and Solution Overview

3.1. Problem Overview

The objective of compilation provenance identification is to deduce, from the to-be-analyzed binary code, the specific compiler-related settings employed during the source code's compilation. This task is viable due to the distinct differences often introduced by the various compilation and optimization settings. These differences significantly influence the field of binary code similarity detection, which in turn highlights the impacts from distinct mechanisms and design choices inherent within different compilers to their compiled binaries, providing evidence of the binaries' compilation provenance. We aim to develop a highly effective compilation provenance identification method that operates at the more fine-grained function level and achieves enhanced detection accuracy in this research, which can be formulated as the following.

Definition 1. *Fine-Grained Compilation Provenance Identification: From an individual binary function f, which is stripped of any debug or symbol information, the diverse compilation settings $\mathcal{D}$ adopted during the compilation processing for producing f can be accurately deduced with a suite of models $\mathcal{M}$ following a learning-based paradigm.*

The "fine-grained" aspect of this definition refers to two meanings. First, compared with existing works [16,29] that perform identification on the whole binary or object file, our subject of analysis is an isolated function, which is independent and devoid of any context such as its adjacent functions in the function call graph. Second, the compilation settings

$\mathcal{D}$ to be identified are more comprehensive, encompassing not only the compiler family but also the optimization level, compiler version, or their combinations, which exceed the simplicity of identifying just the compiler family [27] or optimization level [17] alone.

3.2. Solution Overview

Figure 1 depicts the structural overview of MulCPI, which encapsulates three principal components: the module for intermediate representation preparation, the neural encoding module, and the classification module. To be specific, taking in the binary function to be analyzed, MulCPI firstly derives two distinct intermediate code representations, i.e., the normalized assembly sequence and the attributed control flow graph (ACFG), for it. Subsequently, in the neural encoding module, compilation-indicative features are distilled from the assembly sequence and the ACFG with the Transformer encoder and an attention-augmented GGNN, respectively. These independently collected numerical features are then further aggregated utilizing a self-attention driven fusion schema and are finally fed into an MLP-based classification layer, either for loss computing against the ground-truth label during the model's training phase, or for predicting the compilation provenance details in the detection phase.

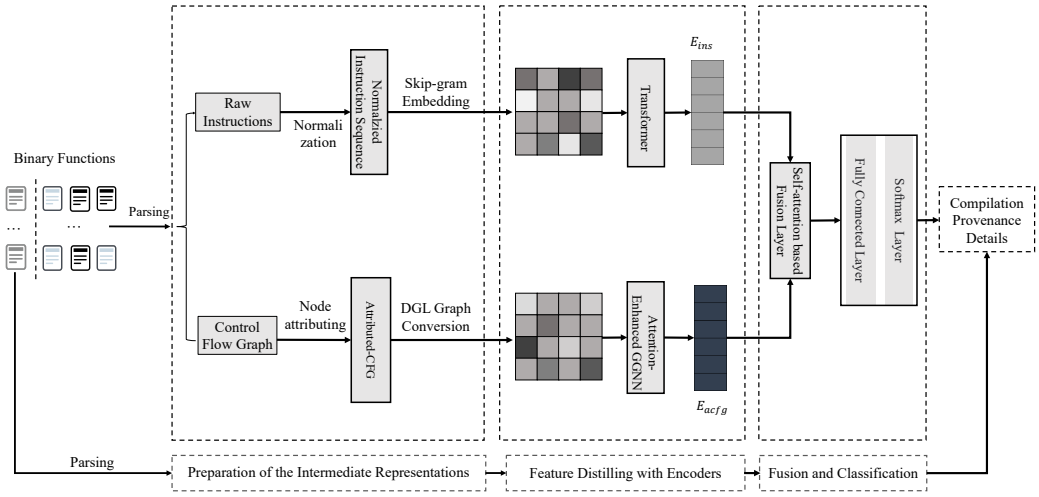

Figure 1. The overall architecture of MulCPI.

4. Core Designs of MulCPI

The following sections present the designing details of MulCPI, including the preparation and processing of the two kinds of intermediate code representation, their corresponding neural encoder for feature distillation, and the strategy for feature fusion from these two distinct intermediate representations.

4.1. Intermediate Representation Preparation

This section details the process of constructing the normalized assembly sequence and the ACFG, on which basis the compilation-significant features are extracted.

4.1.1. Assembly Sequence Normalization

The sequence of assembly instructions has been used as an effective intermediate code representation across a broad spectrum of binary analysis tasks. It has also been established as an effective source in reflecting the impacts of certain compilation configurations on the resultant binary code [4], considering the distinguishable lexical and syntactic features it conveys in the specific contents and the orderings of its comprising assembly instructions. Therewith, MulCPI also regards it as a significant and representative intermediate code representation upon which to operate.

However, as figured out in many prior studies [30,31], direct engagement with the raw assembly instructions suffers non-negligible pitfalls for nuanced binary analysis tasks. MulCPI aims to discern the compilation-specific characteristics rather than functional attributes of the binary code. Direct analysis of the raw assembly instructions risks inundating the process with excessive functional details, thus potentially deteriorating the quality of instruction embeddings due to the retention of a plethora of distinct instructions (which could also exacerbate the OOV problem), while complicating the distillation of compilation-related features during neural encoding. Conversely, overly aggressive normalization of assembly instructions could inject substantial human bias, erasing nuanced yet crucial distinctions, such as those found in the handling of specific predicates by different compilers like GCC and Clang [4].

In consideration of this, we opt for a light-weight abstraction strategy to normalize the assembly instructions. The specific normalization rules applied to each assembly instruction within the sequence are as follows:

- The mnemonic (i.e., the operator) and the operands of the register type within each assembly instruction remain intact.
- The base memory addresses within the operands are replaced uniformly with the symbol *MEM* to bypass non-essential displacement details.
- The immediates with values larger than a predefined threshold (established at 5000 in our implementation) are replaced uniformly with the symbol *IMM*.
- The string literals within each assembly instruction are uniformly replaced with the symbol *STR*.

For instance, with applying the normalization rules, the instruction "add rsp, 8" is normalized to "add rsp, IMM", the assembly instruction "mov ecx, [0x400327b]" is normalized to "mov ecx, MEM", and the instruction "lea rdi, aBinary" becomes "lea rdi, STR".

4.1.2. ACFG Construction

The control flow graph (CFG), with its nodes being the basic blocks and the edges delineating the control flow relationships between them, is a prevalent code intermediate representation in the literature of program analysis. Compared with the assembly instruction sequence, it acts as a more comprehensive intermediate representation that additionally offers the structural aspects of the code apart from the lexical and syntactic aspects conveyed by the specific assembly instructions within each basic block. Thereby, we recognize it as another valuable yet distinct source for extracting compilation indicative features. To leverage this potential, we construct an attributed-CFG on its basis, which serves as a foundation for facilitating the subsequent neural encoding process.

Specifically, given the $CFG = (V, E)$, where V and E denote the set of nodes and edges, respectively, we further associate each node with a set of attributes that well summarize the lexical and syntactic characteristics of its contents. To achieve that, we first normalize the assembly instructions within each basic block with the same abstraction strategy as discussed in Section 4.1.1 and then leverage the efficient sent2vec model to obtain an informative numerical vector.

Consider the basic block BB that corresponds to a node v within in the set V, and let $S = \{w_1, w_2, \cdots, w_n\}$ be its normalized sequence of assembly instructions. Here, w_t signifies the t-th normalized assembly instruction, and n is the count of instructions within BB. We approach each such sequence akin to a sentence, treating each of its normalized instructions as a word within it. Subsequently, we input these 'sentences' derived from our binary dataset into the sent2vec algorithm, employing it to iteratively learn a sentence embedding model $S2V$ in an unsupervised fashion. This model conceptualizes sentence embedding as the mean of the embeddings of its constituent 'words', while also enhancing its learning scope to include both unigrams and n-grams within each 'sentence', thereby av-

eraging out the embeddings of these n-grams along with the individual words. Equation (1) displays the learning objective:

$$\arg\min_{\boldsymbol{U},\boldsymbol{V}} \sum_{\mathcal{S}\in\mathcal{C}} \sum_{w_t\in\mathcal{S}} \left(q_p(w_t)\ell(\boldsymbol{u}_{w_t}^{\top}\boldsymbol{v}_{\mathcal{S}\setminus\{w_t\}}) + |N_{w_t}| \sum_{w'\in V} q_n(w')\ell(-\boldsymbol{u}_{w'}^{\top}\boldsymbol{v}_{\mathcal{S}\setminus\{w_t\}}) \right) \tag{1}$$

where $q_p(w_t)$ is the subsampling likelihood of an instruction w_t being selected to constitute the target positive unigrams of the current assembly sequence S. The function ℓ is the binary logistic loss. $\boldsymbol{u}_w$ represents the target embedding for each instruction w within the vocabulary $\mathcal{V}$, and $\boldsymbol{v}_S$ is the sentence embedding of S, computed by averaging the embeddings of n-grams contained within S. The set N_{w_t} comprises instructions negatively sampled for the instruction w_t within the sequence $\mathcal{S}$, and $q_n(w')$ indicates the probability of an instruction w' in the vocabulary $\mathcal{V}$ being selected to generate the negatives.

With the learnt sentence embedding model, each node v in the CFG can be correlated with a d-dimensional (which is set to 128 in MulCPI) numerical vector v_a, forming an attributed-CFG. Given that MulCPI employs the widely used dgl (https://www.dgl.ai/, accessed on 21 April 2024) library for building up the graph neural network model, Algorithm 1 outlines the process of constructing the ACFG, which is compatible with the graph data structure supported by dgl.

Algorithm 1 Construction of Attributed-CFG Compatible with dgl

Input:
G: CFG of a function
$S2V$: the learnt sent2vec model
Output:
$ACFG$: the attributed control flow graph

```
Edgs = {}
for each edge e in G.edges do
    src, dst = e.src, e.dst
    Edgs.append((src, dst))
end for
ACFG = dgl.graph(Edgs)    ▷ initialize a dgl graph with the edges in the CFG
for each v in G.nodes do
    idx = G.indexof(v)
    S = insNorm(v)    ▷ normalize the instructions within current node
    v_a = S2V(S)    ▷ obtain the numerical attributes of current node with S2V
    ACFG.nodes[idx].data['attr'] = v_a
end for
```

4.2. Feature Gathering through Neural Encoding

This section elaborates on the methodology for extracting features indicative of the compilation details from the normalized assembly sequence and the ACFG intermediate representations, where a Transformer encoder is utilized for the former and an attention-enhanced GGNN is devised for the latter.

4.2.1. Feature Encoding of the Normalized Assembly Sequence

MulCPI extracts informative features from the normalized assembly sequence through a sequence-oriented encoder that follows the Transformer architecture [32], which is renowned for its superior performance over other neural networks like CNNs and LSTMs in encoding sequence-like data. Leveraging the multi-head attention, MulCPI tends to discern not only the subtle lexical and syntactic features interspersed in the assembly instructions but also their long-range contextual relationships.

Specifically, MulCPI first converts the assembly sequence into a $d \times n$-dimensional numerical matrix by adding up the positional embeddings of each normalized instruction

with its word2vec embeddings. Subsequently, this matrix goes through a stack of Transformer blocks, each consisting of a multi-head attention layer, a normalization layer, and a position-wise feed-forward layer, for hidden vector calculation. Finally, MulCPI takes the averagely pooled result of the hidden vectors produced by the top-layer Transformer block as the feature encoding vector, marked as E_{ins}. For MulCPI's specific implementation, the number of Transformer blocks and the heads are set to three and five, respectively.

4.2.2. Feature Encoding of the ACFG

For the ACFG representation as constructed with Algorithm 1 for a binary function, where each node has been independently attributed with the initially collected shallow lexical and syntactic features of the assembly instructions within it, we further devise an attention-augmented graph neural network based on the principles of GGNN [33] to extract informative features. The choice is driven by the network's ability in effectively handling the complex interactions across the nodes through message passing [34,35], with the expectation of distilling higher-level features that comprehensively reflect the impacts of a specific compilation setting on the lexical, syntactic, and structural aspects of its produced binary code. Compared with other prevalent graph neural network architectures like Graph Convolutional Networks (GCN) [36] and Graph Attention Networks (GAT) [37], we improve upon the GGNN structure considering its demonstrated superiority in encoding in-depth features especially in the context of program analysis [38–40].

To be specific, the enhancement is enforced on the message passing process of the original GGNN [33] layer by incorporating an attention mechanism. This allows for a more focused aggregation of interesting information from the neighboring nodes to feature the current node's context, thus enhancing our graph neural encoder's overall ability to highlight relevant features within the graph's context. Formally, we start with a hidden state matrix $H^0 \in \mathbb{R}^{n\times d}$, encapsulating all the d-dimensional real-valued attributes of the graph nodes, and an adjacency matrix $A \in \mathbb{R}^{n\times n}$, delineating the neighborhood connectivity among the n nodes within the ACFG. For each node i in the ACFG, we calculate a hidden vector $\hat{h}_i^{(t+1)}$, which incorporates the information of neighboring nodes through attention scores and their hidden states at previous time step t, with the following equations:

$$\hat{h}_i^{t+1} = \sum_{j\in\mathcal{N}_i} \alpha_j^{t+1} h_j^t \tag{2}$$

$$\alpha_j^{t+1} = \frac{exp(h_j^t W)}{\sum_{j\in\mathcal{N}_i} exp(h_j^t W)} \tag{3}$$

where $\mathcal{N}_i$ signifies the nodes that are contextually connected to node i, encompassing both the node itself and its direct neighbors; α_j^{t+1} denotes the attention score for node j at the time step $t+1$, which is derived by multiplying the previous step hidden states $h_j^t \in H^t$ of node j with a globally shared weight matrix $W \in \mathbb{R}^{d\times 1}$ and normalizing with the Softmax operation.

In essence, the above calculations attempt to use the attention score bundled to each neighbor of the current node to influence how much the neighbor's information affects and is assimilated into the updated state of the given node. Then, the specific computations performed within the GRU unit for obtaining the final hidden states of the node i at time step $t+1$ can be expressed by Equations (4)–(8):

$$h_i^{t+1} = \widetilde{h}_i^{t+1} \odot z^{t+1} + h^t \odot (1 - z^{t+1}) + \hat{h}_i^{t+1} \tag{4}$$

$$\widetilde{h}_i^{t+1} = tanh(W_h a^{t+1} + U_h(r^{t+1} \odot h_i^t) + b_h) \tag{5}$$

$$r^{t+1} = \sigma(W_r a^{t+1} + U_r h_i^t) + b_r) \tag{6}$$

$$z^{t+1} = \sigma(W_z a^{t+1} + U_z h_i^t) + b_z) \tag{7}$$

$$a^{t+1} = \hat{A}^t h_i^t W_a \tag{8}$$

where r and z are the reset and update gate of the original GRU unit, respectively; $\hat{A}$ denotes the Laplacian re-normalized adjacency matrix; the Ws and Us are the weight matrices, which altogether with the bias vectors bs are to be learnt during training; $\odot$ signifies the element-wise multiplication; and σ denotes the sigmoid function.

After obtaining the hidden states of all nodes at the last time step, a graph-level readout layer is appended to derive a comprehensive encoding vector for the entire graph. Specifically, we adopt the max-pooling of all the nodes' hidden states as the feature encoding vector of the ACFG, which can be formulated as Equation (9):

$$E_{acfg} = maxpool\{h_v\}_{v \in ACFG} \tag{9}$$

where h_v denotes the final time step hidden states of a node v in the ACFG. Figure 2 illustrates the whole process of obtaining the E_{acfg} of the ACFG with our devised GGNN enhanced with an attention mechanism.

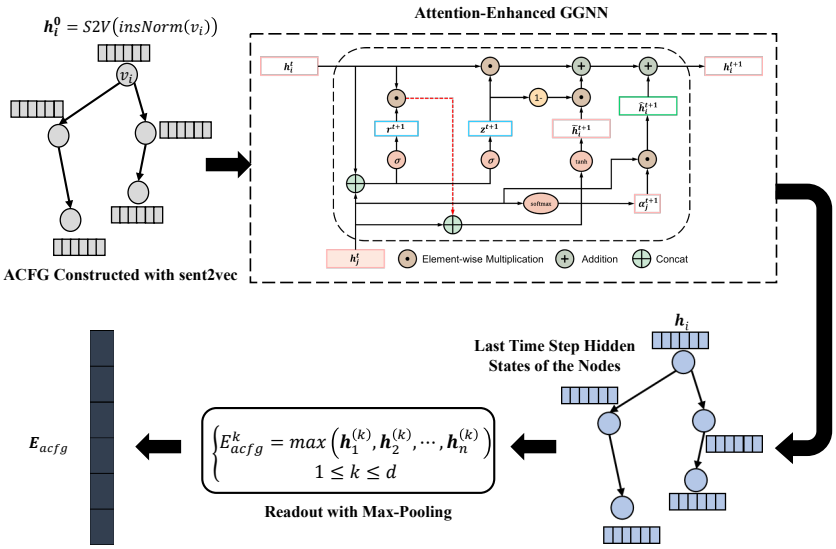

Figure 2. The working flow of MulCPI's feature fusion process.

4.3. Feature Fusion and Classification

After the features indicative of the compilation details have been gathered from the normalized assembly token sequence and the ACFG using the respective neural encoders, here we further mix these separately obtained features to obtain one dense yet perceptive feature vector. Several strategies exist for fusing vectors derived from disparate sources, including direct concatenation, max or mean pooling, and element-wise addition. Given the information provided by the two code intermediate representations, their encoded features could also present a degree of redundancy. Concurrently, the relevance of these features in identifying compilation settings might differ. Hence, to mitigate the challenges posed by redundant features and to discern the most salient ones, we employ a self-attention based fusion strategy to obtain a mixture of the feature vectors in MulCPI.

The fusion strategy is inspired by the encoder–decoder attention layer within the Transformer architecture, in which the self-attention computations are enforced between vectors originating from disparate sources: one sourced from the encoder's output, and the other from the decoder layer's input. This can be adapted to our case directly by regarding the encoded feature vectors gathered from the distinct code intermediate representations as different-sourced data. Figure 3 illustrates the working flow of the fusion process.

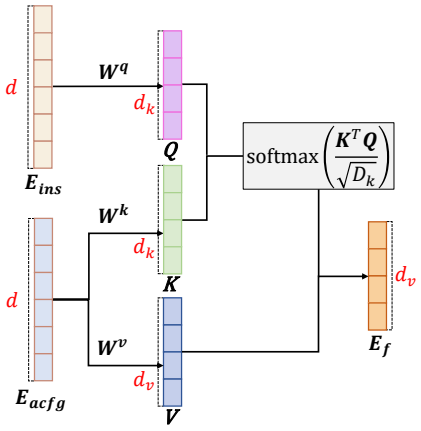

Figure 3. The working flow of MulCPI's feature fusion process.

Specifically, our fusion strategy first derives the query matrix Q from the encoded feature vector $E_{ins} \in \mathbb{R}^d$, as well as derives the key matrix K and the value matrix V from the encoded feature vector $E_{acfg} \in \mathbb{R}^d$, with three distinct linear transformations, respectively.

$$Q = W^q E_{ins},\ K = W^k E_{acfg},\ V = W^v E_{acfg} \tag{10}$$

where W^q, W^k, and W^v denote the learnable weight matrices associated with each linear transformation.

Subsequently, the Q-K-V matrices undergo a standard self-attention calculation to produce a refined output vector $E_f \in \mathbb{R}^{d_v}$, serving as the ultimate feature encoding for function f. This specific calculation can be expressed by Equation (11):

$$E_f = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{11}$$

where d_k denotes the vector dimensionality in Q and K, and d_v signifies that of the vectors in V, which, in order to fulfill the feature concentration goal of the fusion, is intentionally set lower than d.

On this basis, a multilayer perceptron (MLP)-based classification layer, which takes in the function encoding E_f and predicates a detection outcome, is finally appended to constitute a complete compilation provenance identification model.

5. Experiments and Evaluations

5.1. Implementation Details and Dataset Preparation

MulCPI is primarily developed in Python and leverages IDA Pro (https://hex-rays.com/ida-pro/, accessed on 21 April 2024) along with IDAPython (https://github.com/idapython/src, accessed on 21 April 2024) for detailed binary parsing, including the identification of function boundaries, as well as the acquisition of the raw assembly instructions and the original control flow graph for each binary function. The skip-gram and sent2vec algorithms for generating instruction embeddings and for deriving numerical attributes of the CFG nodes, respectively, utilize the implementations sourced from gensim (https://radimrehurek.com/gensim/, accessed on 21 April 2024) and sent2vec (https://github.com/epfml/sent2vec, accessed on 21 April 2024). The Transformer-based neural encoder for extracting features from the normalized instruction sequence, and the attention augmented GGNN encoder for extracting features from the ACFG, are realized using the Pytorch framework in conjunction with the dgl (https://www.dgl.ai/pages/start.html, accessed on 21 April 2024) library. The hyperparameter set pertaining to the neural network components within MulCPI are detailed in Table 1.

Table 1. Hyperparameter Settings of the Neural Network Components within MulCPI.

Neural Component	Parameter	Value
Sent2Vec	the dimensionality of the sentence embedding	128
	the minimal number for the word occurrences	5
	the threshold for sampling	1×10^{-3}
	the max length of word ngram	3
Transformer Encoder	the number of Transformer blocks	3
	the number of self-attention multiheads within each block	5
	hidden state dimensionality of the intermediate layers	256
	hidden state dimensionality of the last layer	128
	dropout rate	0.5
Attention-Enhanced GGNN Encoder	the number of hidden layers	3
	the number of information propagation steps	5
	the node hidden state dimensionality	128
	readout mechanism	max-pooling
	dropout rate	0.5

The publicly available dataset (https://github.com/zztian007/NeuralCI, accessed on 21 April 2024) established by NeuralCI [4] serves as the foundation for evaluating the performance of MulCPI and its comparison with the baseline approaches. This dataset includes diverse 4,810 binaries compiled from 19 extensively utilized C/C++ open-source projects, with the combinations of different compilers in multiple versions, including GCC (4.7, 4.8, 4.9, 5.5, 6.5, and 7.4), Clang (3.8 and 5.0) and ICC 19.0, and their different optimization levels (O0, O1, O2, and O3). For further details on the compilation process, interested readers may refer to NeuralCI, which also discusses the impacts of various compilation settings on the generated binaries.

To derive the function samples, we leveraged IDA Pro to parse each binary file within the base dataset and extracted the assembly instructions and the CFG of each function within the binary using IDAPython. Similarly as in NeuralCI, functions with fewer than 10 instructions were deemed trivial and excluded from analysis, while only functions that contain unique normalized assembly instructions were retained. These resulted in a dataset comprising 854,858 distinct functions, with each of them well labeled according to the compilation settings employed for compiling the binary in which the function resides.

5.2. Experimental Setup

Utilizing these extracted function samples, we conducted different experiments to gauge MulCPI's performance in discerning the diverse facets of compilation provenance, encompassing the identification of compiler family, the specific optimization level, the specific compiler version used, and their combinations. For each specific detection challenge, the function samples with pertinent labels were curated to form the specific dataset for training an identification model and testing its performance. But in all the experiments, the relevant function samples were systematically allocated into train, validation, and test sets, adhering to a consistent ratio of 8:1:1. Moreover, the training of the models all commenced with an initial learning rate of 0.001 and underwent a decay of 0.9 every five epochs, utilizing the Adam optimizer and a batch size of 32. To optimize training efficiency and curtail overfitting, the early stopping schema was set up, which suspended the training upon a lack of improvement in the validation accuracy over five epochs. The model exhibiting the highest validation accuracy was thereafter designated as the definitive detection model and used for subsequent test set performance evaluation. All experimental evaluations were conducted on a server running Ubuntu 12.04 and equipped with 128GB RAM and two NVIDIA RTX3090 GPUs.

Similar to prior research [4,17] on compilation provenance identification, MulCPI adopts four established metrics, including Accuracy, Precision, Recall, and F1-score, for its performance evaluation and comparison with other baseline approaches. Moreover, note

that Accuracy refers to the total accuracy rate, whereas Precision, Recall, and F1-score all represent the weighted averages of precision, recall, and F1-score, respectively, which can be formulated as:

$$\text{Acc.} = \frac{\sum_{i=1}^{k} c_i'}{\sum_{i=1}^{k} c_i}, \quad Prec. = \sum_{i=1}^{k} \frac{c_i}{\sum_{j=1}^{k} c_j} p_i, \quad Rec. = \sum_{i=1}^{k} \frac{c_i}{\sum_{j=1}^{k} c_j} r_i, \quad F1 = \sum_{i=1}^{k} \frac{c_i}{\sum_{j=1}^{k} c_j} f_i \tag{12}$$

where k denotes the number of all possible class label values, $\{c_1, c_2, \cdots, c_k\}$ represent the number of samples corresponding to each class, and $\{c_1', c_2', \cdots, c_k'\}$ denote the number of correctly classified samples for each class by the model; $\{p_1, p_2, \cdots, p_k\}$, $\{r_1, r_2, \cdots, r_k\}$, and $\{f_1, f_2, \cdots, f_k\}$ represent the standard precision, recall, and f1-score calculated for each of the k classes, respectively.

5.3. Experimental Results

In this section, we present the details of the experiments conducted and the findings. The experiments from Sections 5.3.1–5.3.4 evaluate the effectiveness of MulCPI in pinpointing the diverse compilation details, including the compiler family, the optimization level, the specific compiler version, and a combination of them, respectively. Three state-of-the-art deep learning-based methods (NeuralCI$_{cnn}$, NeuralCI$_{gru}$ (https://github.com/zztian007/NeuralCI, accessed on 21 April 2024), and o-glassesX (https://github.com/yotsubo/o-glassesX, accessed on 22 April 2024), as well as three conventional machine learning-based ones (https://github.com/dyninst/toolchain-origin, accessed on 21 April 2024) (Idioms, Graphlets, and ORIGIN), which all support function-level compilation provenance identification, are taken as the benchmark models for a systematic comparison with MulCPI in these sections. Furthermore, in Section 5.3.5, we carry out substitution experiments to evaluate MulCPI's performance by replacing either its neural encoders or the fusion strategy with other alternatives. Finally, an ablation study, which examines the active roles of leveraging multiple distinct intermediate code representations for feature extraction and fusion, as opposed to utilizing solely one kind representation, in conducted in Section 5.3.6.

5.3.1. Performance on Compiler Family Identification

In this study, we consider the compiler family used to compile each function as the ground truth, resulting in a reorganization of all the processed function samples into three categories that are marked with GCC, Clang, and ICC, respectively. Then, the MulCPI model is trained and assessed against other benchmark methods by adhering to the experimental settings as outlined in Section 5.2.

As summarized in Table 2, MulCPI demonstrates excellent detection capability in the compiler family identification task. It achieves an Accuracy of 0.993 and an F1-score of 0.992, surpassing all comparison methods in both metrics. This indicates MulCPI's remarkable ability in distilling insightful features from the compiled binaries. Additionally, it is noteworthy that DL-based methods consistently outperformed the traditional ML-based approaches in terms of all the performance metrics, highlighting the superior feature extraction capability of deep neural encoders.

Table 2. Performance of MulCPI against the benchmark methods on compiler family identification.

Metric	MulCPI	NeuralCI$_{cnn}$	NeuralCI$_{gru}$	o-glassesX	Idioms	Graphlets	ORIGIN
Acc.	0.993	0.985	0.987	0.988	0.939	0.842	0.940
Prec.	0.995	0.985	0.987	0.989	0.943	0.845	0.944
Rec.	0.991	0.985	0.987	0.986	0.939	0.842	0.940
F_1	0.992	0.985	0.987	0.985	0.938	0.838	0.939

5.3.2. Performance on Optimization Level Identification

This experiment assessed the effectiveness of MulCPI against the comparison methods in pinpointing the specific optimization level utilized during the compilation process. Thus,

the optimization levels with respect to each certain compiler served as the ground-truth labels to obtain all the function samples reorganized into the corresponding categories. As pointed out by previous research findings [3,4], distinguishing between binaries compiled with the O2 and O3 optimization levels is challenging. Thereby, the same as the settings in previous works [3,4,15], we simplified this task by reducing the four-level optimization options to two abstracted categories O_L and O_H. Specifically, the optimization levels O0 and O1, in which relatively fewer code optimizations are applied, were grouped into O_L, while levels O2 and O3, characterized by more extensive and aggressive optimizations, are aggregated into O_H. Furthermore, the tripartite classification scheme [4] which condenses the optimization levels into O0, O1, and O_H, as well as the original four-level optimization settings, are also explored in this experiment to offer a nuanced perspective on the model's performance across varied optimization level identification scenarios.

The outcomes of the evaluation are detailed in Table 3, where the achieved higher metric values affirm MulCPI's superior performance over the benchmark methods across all the metrics. Notably, in the two-level optimization identification task, MulCPI attains a mean accuracy and F1-score of 0.967, surpassing the competing approaches with the max improvements of 4.96% and 4.56%, and on average by 2.56% and 2.63% for Accuracy and F1-scores, respectively. For the case of the three-level optimization identification challenge, it achieves a mean accuracy and F1-score of 0.964, exceeding the performance of other models by a maximum of 7.34% and 7.63%, and an average of 2.86% and 3.05% in Accuracy and F1-scores, respectively. These results highlight MulCPI's superiority in more comprehensively and precisely discerning the significant features of compilation encoded within fused real-valued vectors.

Furthermore, it is noteworthy that MulCPI's performance in pinpointing optimization levels varies across different compilers, with the most and least successful outcomes reported for GCC and Clang, respectively, reflecting approximately a 6.9% gap in Accuracy. This implies that the nuances introduced by GCC's optimization levels in its binaries are more readily identifiable, contrasting with the more challenging task of discerning optimization levels in binaries compiled by Clang. Such differences in identification challenges further point to the distinct approaches compilers take in specifying their optimization levels.

Table 3. Performance comparison results on optimization level identification.

Level Settings	Model	GCC				Clang				ICC			
		Acc.	**Prec.**	**Rec.**	**F1**	**Acc.**	**Prec.**	**Rec.**	**F1**	**Acc.**	**Prec.**	**Rec.**	**F1**
2-Levels (O_L, O_H)	MulCPI	0.991	0.991	0.991	0.991	0.932	0.933	0.932	0.932	0.978	0.978	0.978	0.978
	NeuralCI$_{cnn}$	0.987	0.987	0.987	0.987	0.918	0.918	0.918	0.918	0.957	0.957	0.957	0.957
	NeuralCI$_{gru}$	0.988	0.988	0.988	0.988	0.912	0.913	0.912	0.913	0.953	0.954	0.953	0.953
	o-glassesX	0.986	0.986	0.986	0.986	0.920	0.921	0.918	0.919	0.973	0.973	0.972	0.973
	Idioms	0.970	0.971	0.970	0.970	0.865	0.864	0.865	0.855	0.972	0.972	0.972	0.972
	Graphlets	0.940	0.941	0.940	0.941	0.858	0.854	0.858	0.849	0.970	0.970	0.970	0.970
	ORIGIN	0.972	0.973	0.972	0.972	0.871	0.870	0.871	0.862	0.976	0.976	0.976	0.976
3-Levels (O_0, O_1, O_H)	MulCPI	0.993	0.993	0.993	0.993	0.924	0.925	0.924	0.924	0.975	0.975	0.975	0.975
	NeuralCI$_{cnn}$	0.989	0.989	0.989	0.989	0.910	0.911	0.910	0.911	0.960	0.960	0.960	0.960
	NeuralCI$_{gru}$	0.988	0.989	0.988	0.988	0.913	0.916	0.913	0.914	0.949	0.949	0.949	0.949
	o-glassesX	0.989	0.989	0.989	0.989	0.916	0.917	0.916	0.916	0.971	0.971	0.970	0.971
	Idioms	0.966	0.967	0.966	0.966	0.857	0.862	0.857	0.849	0.970	0.970	0.970	0.969
	Graphlets	0.920	0.922	0.920	0.920	0.825	0.829	0.825	0.821	0.953	0.952	0.952	0.952
	ORIGIN	0.968	0.969	0.968	0.968	0.864	0.868	0.864	0.858	0.974	0.974	0.974	0.974

5.3.3. Performance on Compiler Version Identification

This section explores the capability of MulCPI in revealing from the binary code the exact version of the compiler used for compiling it, which intuitively is a more intricate and challenging task. Therewith, the function samples compiled with either GCC or Clang, where multiple versions of them have been provided for constructing the dataset, were used to train the models and assess their performances.

As shown in the metric values in Table 4, MulCPI exhibits leading performance over all the benchmark methods, with the average improvements of 5.48% on Accuracy and 5.81% on F1-score, respectively. This again indicates the subtle feature aware ability of MulCPI attributed to its correct choice of leveraging multiple intermediate representations and the elaborately designed neural encoders for capturing the more comprehensive lexical, syntactical, and structural code aspects. Moreover, varied detection efficacy is observed across different compilers. As the metric values show, consistently higher accuracies were achieved for all the models in identifying the major versions of GCC over Clang. These findings indicate the differences in code variation between compiler versions, with GCC demonstrating a more pronounced divergence in the compiled code produced with its different major releases compared to Clang. Additionally, significantly much lower Accuracy rates were obtained in discerning between the minor GCC versions (4.7, 4.8, and 4.9) compared to those achieved in identifying major versions. This aligns with the common sense that major version updates of compilers tend to introduce more substantial and breaking changes than minor updates.

Table 4. Performance comparison results on compiler version identification.

Model	GCC (Major)				Clang				GCC (Minor)			
	Acc.	**Prec.**	**Rec.**	**F1**	**Acc.**	**Prec.**	**Rec.**	**F1**	**Acc.**	**Prec.**	**Rec.**	**F1**
MulCPI	0.956	0.955	0.956	0.955	0.858	0.858	0.856	0.857	0.792	0.799	0.791	0.793
NeuralCI$_{cnn}$	0.944	0.943	0.944	0.943	0.841	0.841	0.841	0.839	0.770	0.774	0.770	0.771
NeuralCI$_{gru}$	0.940	0.940	0.940	0.940	0.823	0.821	0.823	0.821	0.754	0.754	0.754	0.754
o-glassesX	0.948	0.947	0.948	0.947	0.851	0.848	0.851	0.849	0.786	0.788	0.786	0.787
Idioms	0.892	0.895	0.892	0.890	0.819	0.827	0.819	0.806	0.740	0.770	0.740	0.741
Graphlets	0.855	0.858	0.855	0.852	0.780	0.775	0.780	0.762	0.629	0.635	0.629	0.630
ORIGIN	0.905	0.908	0.905	0.903	0.838	0.845	0.838	0.828	0.759	0.778	0.759	0.760

5.3.4. Performance on Compilation Setting Combination Identification

Deciphering the combination of compilation settings employed to compile a binary typically involves the synergistic use of multiple models, with each unraveling a specific aspect of these settings. For instance, to ascertain both the compiler family and the optimization level utilized to produce the binary function, one might initially deploy a model learnt specifically for identifying the compiler family only. Subsequently, a specialized model for optimization level identification but which is tailored to the previously identified compiler family could be employed to determine the optimization level. Alternatively, learning a unified model capable of simultaneously detecting all these settings tends to be a more attractive and preferable approach. To explore the potential of MulCPI in application to such a more complex task, we examined its ability to simultaneously unveil these compilation details from a binary function.

Table 5 presents the experimental results (the same as in NeuralCI [4], only the major compiler versions are involved in this experiment), where the performance of the models are separately evaluated and reported for the 2-level and the 3-level optimization level settings in Table 5a,b, respectively. As the results show, the detection accuracy values and F1-scores of all the models are notably lower compared to those achieved in less challenging identification tasks discussed in the previous sections. Yet, MulCPI still behaves the best among all the models in terms of each performance metric. Its detection accuracies reach 0.855 and 0.853 under the 2-level and 3-level optimization settings, surpassing the other methods with an average improvement of 12.0% and 12.1%, and a maximum improvement of 32.8% and 35.0%, respectively.

Table 5. Performance comparison results on compilation setting combination identification.

(a) Results for the 2-Level Optimization Level Setting							
Metric	MulCPI	NeuralCI$_{cnn}$	NeuralCI$_{gru}$	o-glassesX	Idioms	Graphlets	ORIGIN
Acc.	0.855	0.830	0.832	0.842	0.731	0.644	0.742
Prec.	0.856	0.830	0.833	0.843	0.750	0.661	0.761
Rec.	0.855	0.830	0.832	0.841	0.731	0.644	0.742
F_1	0.853	0.828	0.832	0.839	0.724	0.634	0.736
(b) Results for the 3-Level Optimization Level Setting							
Metric	MulCPI	NeuralCI$_{cnn}$	NeuralCI$_{gru}$	o-glassesX	Idioms	Graphlets	ORIGIN
Acc.	0.853	0.837	0.827	0.846	0.729	0.632	0.744
Prec.	0.856	0.840	0.827	0.848	0.757	0.654	0.771
Rec.	0.852	0.837	0.827	0.843	0.729	0.632	0.744
F_1	0.851	0.833	0.826	0.841	0.726	0.629	0.744

As illustrated by the values in Table 5, MulCPI wins over all the baseline methods consistently in terms of all the performance metrics. To further establish whether these differences in performance are statistically significant, we conducted the Wilcoxon rank sum test and t-tests between MulCPI and each baseline method, utilizing a 5 × 2 cross-validation setting. The resulting *p*-values for Accuracy and the F1-score are detailed in Table 6 Notably, all the *p*-values are below the 0.05 threshold for either test, confirming a statistically significant difference in performance between MulCPI and the baseline methods on the challenging task of compilation setting combination identification.

Table 6. Statistical significance testing between MulCPI and the baseline methods.

Level Setting	Method Pair	Wilcoxon Rank Sum		T-Test	
		Acc. *p*-Value	F_1	Acc. *p*-Value	F_1 *p*-Value
2-Levels (O_L, O_H)	MulCPI vs. NeuralCI$_{cnn}$	0.0148	0.0151	0.0100	0.0143
	MulCPI vs. NeuralCI$_{gru}$	0.0153	0.0155	0.0121	0.0137
	MulCPI vs. o-glassesX	0.0182	0.0156	0.0167	0.0142
	MulCPI vs. Idioms	0.0003	0.0003	0.0001	0.0001
	MulCPI vs. Graphlets	0.0001	0.0001	0.0003	0.0003
	MulCPI vs. ORIGIN	0.0004	0.0003	0.0001	0.0001
3-Levels (O_0, O_1, O_H)	MulCPI vs. NeuralCI$_{cnn}$	0.0210	0.0173	0.0149	0.0106
	MulCPI vs. NeuralCI$_{gru}$	0.0110	0.0103	0.0090	0.0090
	MulCPI vs. o-glassesX	0.0256	0.0187	0.0173	0.0125
	MulCPI vs. Idioms	0.0003	0.0003	0.0001	0.0001
	MulCPI vs. Graphlets	0.0001	0.0001	0.0002	0.0002
	MulCPI vs. ORIGIN	0.0004	0.0003	0.0001	0.0001

5.3.5. Substitutional Study

In this experiment, we performed alternative experiments by substituting the encoders originally utilized in MulCPI for feature extraction with other widely recognized neural network structures. To be specific, we incorporated three other sequence-oriented models—TextCNN, BiLSTM, and DPCNN—to process the normalized instruction sequences, in addition to the Transformer encoder originally used by MulCPI. For feature extraction from the ACFG, we considered two other well-known graph-oriented neural encoders, GCN and GAT, as alternatives to our attention-enhanced GGNN, to evaluate their performance comparatively.

The evaluation results of each identification task are depicted in Table 7. It should be noted that the metric values for the optimization level identification task refer to the results obtained under the 3-level identification scenario, considering the nearly identical performance to the 2-level identification scenario as observed in Tables 3 and 4. Moreover, the metric values were averaged across the different compilers to offer an overall view of

the models' performance in both the optimization level identification and the major version identification tasks. The experimental findings reveal that MulCPI's blend using of the neural encoders makes it the most effective model for compiler provenance identification, where substituting its components with the aforementioned alternatives generally results in varied degrees of performance decline.

Table 7. Performance of MulCPI with alternative deep neural encoders.

Task	Metric	MulCPI	MulCPI$_{textcnn}$	MulCPI$_{bilstm}$	MulCPI$_{dpcnn}$	MulCPI$_{gcn}$	MulCPI$_{gat}$
Compiler Family	Acc.	0.993	0.988	0.989	0.991	0.986	0.990
	Prec.	0.995	0.989	0.991	0.992	0.988	0.991
	Rec.	0.991	0.986	0.987	0.989	0.983	0.989
	F_1	0.992	0.987	0.988	0.990	0.984	0.990
Optimization Level	Acc.	0.964	0.958	0.959	0.962	0.949	0.956
	Prec.	0.964	0.959	0.957	0.963	0.948	0.956
	Rec.	0.964	0.958	0.959	0.961	0.948	0.955
	F_1	0.964	0.958	0.959	0.962	0.948	0.955
Major Version	Acc.	0.907	0.903	0.902	0.905	0.893	0.904
	Prec.	0.907	0.903	0.902	0.904	0.892	0.904
	Rec.	0.906	0.903	0.903	0.905	0.893	0.904
	F_1	0.906	0.903	0.903	0.905	0.893	0.904
Minor Version	Acc.	0.792	0.788	0.789	0.790	0.763	0.788
	Prec.	0.799	0.793	0.793	0.794	0.767	0.792
	Rec.	0.791	0.785	0.787	0.787	0.762	0.786
	F_1	0.793	0.788	0.790	0.791	0.764	0.788
Setting Combination	Acc.	0.853	0.850	0.849	0.852	0.842	0.851
	Prec.	0.856	0.854	0.853	0.855	0.844	0.853
	Rec.	0.852	0.849	0.849	0.852	0.842	0.851
	F_1	0.851	0.847	0.846	0.850	0.839	0.848

5.3.6. RQ4: Ablation Study

For this experiment, an ablation study regarding the effects of each intermediate code representation was carried out to investigate whether considering the feature vectors distilled from both of them indeed help boost MulCPI's capability in pinpointing the compilation provenance details, over relying on a singular feature vector extracted from one representation alone. Therewith, we disabled the feature extraction component within MulCPI that corresponded to the handling of certain code intermediate representation, and then evaluated and compared the performance of these pruned models with the complete MulCPI. The efficacy of our devised attention-enhanced GGNN encoder for feature extraction from the ACFG was also evaluated by peeling off the attention mechanism blended in it. For simplicity, we refer to the simplified versions of MulCPI operating merely on the normalized instruction sequence or the ACFG as MulCPI$_{ins}$ and MulCPI$_{acfg}$, respectively, and refer to the model adopting the original GGNN as MulCPI$_{no_att}$.

As exhibited by the experimental results in Table 8, the complete MulCPI model outperforms its simplified counterparts, which rely solely on the single vector derived from one intermediate code representation. This enhancement in performance, achieved by leveraging multiple distinct intermediate representations for feature extraction, underscores the importance and advantage of considering various code aspects. This also suggest that models trained with merely one certain code representation tend to struggle to comprehensively capture the nuanced features present in the code, leading to poorer identification capability. Particularly, for the challenging compilation setting combination identification task, there is a noticeable decline in Accuracy, which drops by 3.28%, and F1-score decreases by 3.41%, when stripping off features collected from the ACFG and only utilizing features from the normalized assembly sequence. This decline can be attributed to this particular code representation's limitation of majorly manifesting the lexical and syntactic characteristics of code, while the compilation optimizations could also alter the binary code's structural aspects to a large extent. It can also be inferred that there are some overlaps in the features extracted from the normalized assembly instruction sequence

and the ACFG. However, identifying and integrating the unique features from these two different sources help improve the model's identification capability. Furthermore, the reduced performance of $MulCPI_{no\_att}$ indicates that our enhancement of the GGNN with an attention mechanism allows the encoder to identify more subtle yet significant features that may have been overlooked by the original GGNN.

Table 8. Ablation study results with disabled certain core designs in MulCPI.

Task	Metric	MulCPI	MulCPI$_{ins}$	MulCPI$_{acfg}$	MulCPI$_{no_attt}$
Compiler Family	Acc.	0.993	0.982	0.988	0.984
	Prec.	0.995	0.982	0.990	0.987
	Rec.	0.991	0.980	0.986	0.983
	F_1	0.992	0.981	0.987	0.984
Optimization Level	Acc.	0.964	0.947	0.957	0.948
	Prec.	0.964	0.947	0.957	0.948
	Rec.	0.964	0.946	0.957	0.947
	F_1	0.964	0.947	0.957	0.947
Major Version	Acc.	0.907	0.888	0.898	0.885
	Prec.	0.907	0.888	0.898	0.886
	Rec.	0.906	0.889	0.898	0.885
	F_1	0.906	0.888	0.898	0.885
Minor Version	Acc.	0.792	0.757	0.783	0.764
	Prec.	0.799	0.759	0.786	0.768
	Rec.	0.791	0.756	0.782	0.761
	F_1	0.793	0.758	0.783	0.764
Setting Combination	Acc.	0.853	0.825	0.845	0.833
	Prec.	0.856	0.828	0.847	0.835
	Rec.	0.852	0.823	0.843	0.831
	F_1	0.851	0.822	0.841	0.829

6. Discussion

Based on the above experimental evaluations conducted, it can be observed that the difficulty of training an effective compilation provenance identification model varies significantly under different identification challenges, suggesting that different compilation settings preserve varying amounts of distinctive clues in the final binaries. More specifically, as illustrated by the experimental results, regardless of the specific methods (i.e., the conventional machine learning-based or the SOTA deep learning-based ones) adopted, the difficulty of learning a model that accurately works for compilation provenance identification generally increases in the order of $\mathcal{M}_f < \mathcal{M}_o < \mathcal{M}_v < \mathcal{M}_{combo}$, where $\mathcal{M}_f, \mathcal{M}_o, \mathcal{M}_v$ signify the models learnt for the detection of compiler family, optimization level, and compiler version, respectively, while $\mathcal{M}_{combo}$ refers to the model for unveiling the combinations of compilation setting.

6.1. Threats to Validity

Just like many other binary analysis studies [4,10,41] focusing on individual functions, MulCPI operates under the assumption that the function under examination can always be accurately identified and extracted from the binary file. In our study, we leverage IDA Pro, the leading binary reverse engineering tool, to alleviate such issues of inaccurate function sample acquisition, considering its high performance and prevalence in binary analysis. While other tools like Binary Ninja, Ghidra, and Angr are also compatible with MulCPI, accurately parsing binaries remains an open challenge [42,43]. MulCPI could potentially benefit from the emerging deep learning-driven binary disassembly techniques [44–46] to enhance reliability in the acquisition of functions.

MulCPI's effectiveness may be hindered by the diverse code obfuscation strategies [47,48] applied to the code under analysis. Techniques such as compression and

encryption can disrupt the initial analysis phase by preventing accurate function parsing and extraction, which are critical for MulCPI and other function-level analysis methods. Furthermore, obfuscation techniques like instruction replacement and dead code insertion, which alter the code, could obscure or eliminate key features vital for compilation provenance identification, reducing the detection accuracy. Addressing these challenges may involve initially deobfuscating [49,50] the binaries or applying adversarial training [51,52], which involves training the compilation provenance identification model on obfuscated samples to enhance its resilience against obfuscation tactics. Tackling robust compilation provenance identification against the code obfuscation countermeasures presents a significant research challenge that we aim to explore as a future work.

6.2. Limitations

As exhibited in the experimental evaluations, especially the substitutional and ablation study results, the carefully picked and elaborately designed neural encoders, as well as the more comprehensive lexical, syntactical, and structural feature fusion from multiple distinct intermediate representations, endow MulCPI with a heightened capability to discern the nuanced yet pivotal features, helping it outperform other methods under the various compilation provenance identification tasks. However, the collection and incorporation of features from multiple intermediate code representations inevitably escalate computational overhead, placing MulCPI in an inferior position in terms of the runtime efficiency compared with the methods operating on a single code representation.

Currently, MulCPI has only been evaluated on the binary functions of X64, one of the most represented and widely used instruction architecture sets (ISAs). Theoretically speaking, our method is not restricted to any specific ISA, as training with abundant binary function samples of the corresponding ISA can always obtain an identification model. However, there generally are significant morphological and syntactic gaps between different ISAs; thus, the instruction normalization rules and the neural encoders designed for X64 binary functions may not translate well to other ISAs, such as ARM64. Moreover, as in the many studies targeting architecture agnostic binary code similarity detection [53–55], achieving ISA agnostic compilation provenance identification is also highly appealing. We leave the exploration of adapting MulCPI to different ISAs and cross-ISA compilation provenance identification as interesting future works.

7. Conclusions

Targeting the challenging yet important compilation provenance identification problem, we present an innovative and high-performing method called MulCPI for fine-grained compilation provenance identification on individual functions. It follows the cutting-edge deep learning paradigm, but different from existing methods which generally operate on a single intermediate code representation, it jointly leverages carefully devised sequence-oriented and graph-oriented neural encoders to more comprehensively capture and fuse the diverse significant features that reflect the rich lexical, syntactic, and structural aspects of the binary functions. The extensive experiments conducted on a public dataset validate MulCPI's remarkable capability in capturing the compilation-indicative features and its superior performance over existing methods in identifying the various compilation aspects. The ablation study also confirms the indispensable role of MulCPI's core components, notably the efficacy of the devised attention-enhanced gated graph neural network encoder and the strategic incorporation of multiple code representations.

Author Contributions: Conceptualization, Y.G.; Methodology, Y.G. and L.L.; Validation, Y.W.; Investigation, Y.L.; Data curation, Y.L.; Writing—original draft, L.L.; Writing—review & editing, R.L. and Y.W.; Supervision, Y.G.; Funding acquisition, R.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded in part by the Key Research Project of Shaanxi Higher Education Teaching Reform (23JG001), and the Research Project of Higher Education Science of the China Association of Higher Education (22ZXKS0308).

Data Availability Statement: The data presented in this study are openly available at https://github.com/gyjuice/MulCPI (accessed on 21 April 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Ren, X.; Ho, M.; Ming, J.; Lei, Y.; Li, L. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual, Canada, 20–25 June 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 142–157, PLDI 2021. [CrossRef]
2. Tian, Z.; Liu, T.; Zheng, Q.; Zhuang, E.; Fan, M.; Yang, Z. Reviving sequential program birthmarking for multithreaded software plagiarism detection. *IEEE Trans. Softw. Eng.* **2017**, *44*, 491–511. [CrossRef]
3. Rosenblum, N.; Miller, B.P.; Zhu, X. Recovering the toolchain provenance of binary code. In Proceedings of the International Symposium on Software Testing and Analysis, Toronto, ON, Canada, 17–21 July 2011; pp. 100–110.
4. Tian, Z.; Huang, Y.; Xie, B.; Chen, Y.; Chen, L.; Wu, D. Fine-Grained Compiler Identification With Sequence-Oriented Neural Modeling. *IEEE Access* **2021**, *9*, 49160–49175. [CrossRef]
5. He, X.; Wang, S.; Xing, Y.; Feng, P.; Wang, H.; Li, Q.; Chen, S.; Sun, K. BinProv: Binary Code Provenance Identification without Disassembly. In Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses, Limassol, Cyprus, 26–28 October 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 350–363. RAID '22. [CrossRef]
6. Jang, H.; Murodova, N.; Koo, H. ToolPhet: Inference of Compiler Provenance from Stripped Binaries with Emerging Compilation Toolchains. *IEEE Access* **2024**, *12*, 12667–12682. [CrossRef]
7. Otsubo, Y.; Otsuka, A.; Mimura, M. Compiler Provenance Recovery for Multi-CPU Architectures Using a Centrifuge Mechanism. *IEEE Access* **2024**, *12*, 34477–34488. [CrossRef]
8. Du, Y.; Alrawi, O.; Snow, K.; Antonakakis, M.; Monrose, F. Improving Security Tasks Using Compiler Provenance Information Recovered At the Binary-Level. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, Melbourne, Australia, 10–14 July 2023; Association for Computing Machinery: New York, NY, USA, 2023; pp. 2695–2709, CCS '23. [CrossRef]
9. Pei, K.; Xuan, Z.; Yang, J.; Jana, S.; Ray, B. Learning Approximate Execution Semantics From Traces for Binary Function Similarity. *IEEE Trans. Softw. Eng.* **2023**, *49*, 2776–2790. [CrossRef]
10. Qasem, A.; Debbabi, M.; Lebel, B.; Kassouf, M. 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, Melbourne, Australia, 10–14 July 2023; Association for Computing Machinery: New York, NY, USA, 2023; pp. 443–456, ASIA CCS '23. [CrossRef]
11. Tian, Z.; Zheng, Q.; Liu, T.; Fan, M.; Zhuang, E.; Yang, Z. Software Plagiarism Detection with Birthmarks Based on Dynamic Key Instruction Sequences. *IEEE Trans. Softw. Eng.* **2015**, *41*, 1217–1235. [CrossRef]
12. Du, Y.; Court, R.; Snow, K.; Monrose, F. Automatic Recovery of Fine-grained Compiler Artifacts at the Binary Level. In Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC 22), Carlsbad, CA, USA, 11–13 July 2022; pp. 853–868.
13. Kalgutkar, V.; Kaur, R.; Gonzalez, H.; Stakhanova, N.; Matyukhina, A. Code Authorship Attribution: Methods and Challenges. *ACM Comput. Surv.* **2019**, *52*, 1–36. [CrossRef]
14. Rosenblum, N.E.; Miller, B.P.; Zhu, X. Extracting compiler provenance from program binaries. In Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Toronto, ON, Canada, 20 June 2010; pp. 21–28.
15. Rahimian, A.; Shirani, P.; Alrbaee, S.; Wang, L.; Debbabi, M. Bincomp: A stratified approach to compiler provenance attribution. *Digit. Investig.* **2015**, *14*, S146–S155. [CrossRef]
16. Yang, S.; Shi, Z.; Zhang, G.; Li, M.; Ma, Y.; Sun, L. Understand Code Style: Efficient CNN-Based Compiler Optimization Recognition System. In Proceedings of the IEEE International Conference on Communications, Shanghai, China, 22–24 May 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 1–6.
17. Chen, Y.; Shi, Z.; Li, H.; Zhao, W.; Liu, Y.; Qiao, Y. Himalia: Recovering compiler optimization levels from binaries by deep learning. In Proceedings of the SAI Intelligent Systems Conference, London, UK, 6–7 September 2018; Springer: Cham, Switzerland, 2018; pp. 35–47.
18. Ji, Y.; Cui, L.; Huang, H.H. Vestige: Identifying Binary Code Provenance for Vulnerability Detection. In Proceedings of the Applied Cryptography and Network Security, Kamakura, Japan, 21–24 June 2021; Sako, K., Tippenhauer, N.O., Eds.; Springer: Cham, Switzerland, 2021; pp. 287–310.
19. Otsubo, Y.; Otsuka, A.; Mimura, M.; Sakaki, T.; Ukegawa, H. o-glassesX: Compiler Provenance Recovery with Attention Mechanism from a Short Code Fragment. In Proceedings of the 2020 Workshop on Binary Analysis Research, San Diego, CA, USA, 23 February 2020.

20. Kim, J.; Genkin, D.; Leach, K. Revisiting Lightweight Compiler Provenance Recovery on ARM Binaries. In Proceedings of the 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), Melbourne, Australia, 15–16 May 2023; pp. 292–303. [CrossRef]
21. Austin, T.H.; Filiol, E.; Josse, S.; Stamp, M. Exploring hidden markov models for virus analysis: A semantic approach. In Proceedings of the Hawaii International Conference on System Sciences, Wailea, HI, USA, 7–10 January 2013; IEEE: Piscataway, NJ, USA, 2013; pp. 5039–5048.
22. Toderici, A.H.; Stamp, M. Chi-squared distance and metamorphic virus detection. *J. Comput. Virol. Hacking Tech.* **2013**, *9*, 1–14. [CrossRef]
23. Tian, Z.; Tian, B.; Lv, J.; Chen, Y.; Chen, L. Enhancing vulnerability detection via AST decomposition and neural sub-tree encoding. *Expert Syst. Appl.* **2024**, *238*, 121865. [CrossRef]
24. Yan, Y.; Feng, Y.; Fan, H.; Xu, B. DLInfer: Deep Learning with Static Slicing for Python Type Inference. In Proceedings of the 45th International Conference on Software Engineering, Melbourne, Australia, 14–20 May 2023; IEEE Press: Piscataway, NJ, USA, 2023; pp. 2009–2021, ICSE '23. [CrossRef]
25. Tian, Z.; Tian, J.; Wang, Z.; Chen, Y.; Xia, H.; Chen, L. Landscape estimation of solidity version usage on Ethereum via version identification. *Int. J. Intell. Syst.* **2022**, *37*, 450–477. [CrossRef]
26. Pizzolotto, D.; Inoue, K. Identifying Compiler and Optimization Options from Binary Code using Deep Learning Approaches. In Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), Adelaide, Australia, 28 September–2 October2020; pp. 232–242. [CrossRef]
27. Luca, M.; Giuseppe, A.D.L.; Fabio, P.; Leonardo, Q.; Roberto, B. Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. In Proceedings of the 2019 Workshop on Binary Analysis Research (BAR), San Diego, CA, USA, 24 February 2019; pp. 1–11.
28. Benoit, T.; Marion, J.Y.; Bardin, S. Binary level toolchain provenance identification with graph neural networks. In Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 9–12 March 2021; pp. 131–141. [CrossRef]
29. IDA Pro. Available online: https://hex-rays.com/ida-pro/ (accessed on 21 April 2024).
30. Li, X.; Qu, Y.; Yin, H. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, 15–19 November 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 3236–3251, CCS '21. [CrossRef]
31. Wang, H.; Qu, W.; Katz, G.; Zhu, W.; Gao, Z.; Qiu, H.; Zhuge, J.; Zhang, C. jTrans: Jump-aware transformer for binary code similarity detection. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual, 18–22 July 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 1–13, ISSTA 2022. [CrossRef]
32. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.u.; Polosukhin, I. Attention is All you Need. In Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017; Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R., Eds.; Curran Associates, Inc.: Hook, NY, USA, 2017; Volume 30.
33. Li, Y.; Zemel, R.; Brockschmidt, M.; Tarlow, D. Gated Graph Sequence Neural Networks. In Proceedings of the ICLR '16, Proceedings of ICLR '16, San Juan, Puerto Rico, 2–4 May 2016.
34. Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; Yu, P.S. A Comprehensive Survey on Graph Neural Networks. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *32*, 4–24. [CrossRef] [PubMed]
35. Wu, L.; Cui, P.; Pei, J.; Zhao, L.; Guo, X. Graph Neural Networks: Foundation, Frontiers and Applications. In Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, 14–18 August 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 4840–4841, KDD '22. [CrossRef]
36. Schlichtkrull, M.; Kipf, T.N.; Bloem, P.; vanaden Berg, R.; Titov, I.; Welling, M. Modeling Relational Data with Graph Convolutional Networks. In Proceedings of the The Semantic Web, Monterey CA, USA, 8–12 October 2018; Gangemi, A., Navigli, R., Vidal, M.E., Hitzler, P., Troncy, R., Hollink, L., Tordai, A., Alam, M., Eds.; Springer: Cham, Switzerland, 2018; pp. 593–607.
37. Velickovic, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; Bengio, Y. Graph Attention Networks. In Proceedings of the International Conference on Learning Representations, Vancouver, BC, Canada, 30 April–3 May 2018.
38. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In Proceedings of the Advances in Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019; Wallach, H., Larochelle, H., Beygelzimer, A., Alche-Buc, F., Fox, E., Garnett, R., Eds.; Curran Associates, Inc.: Hook, NY, USA, 2019; Volume 32.
39. Dinella, E.; Dai, H.; Li, Z.; Naik, M.; Song, L.; Wang, K. Hoppity: Learning graph transformations to detect and fix bugs in programs. In Proceedings of the International Conference on Learning Representations, Addis Ababa, Ethiopia, 30 April 2020.
40. Ding, Y.; Suneja, S.; Zheng, Y.; Laredo, J.; Morari, A.; Kaiser, G.; Ray, B. VELVET: A noVel Ensemble Learning approach to automatically locate VulnErable sTatements. In Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 15–18 March 2022; pp. 959–970. [CrossRef]
41. Tian, Z.; Mao, H.; Huang, Y.; Tian, J.; Li, J. Fine-Grained Obfuscation Scheme Recognition on Binary Code. In Proceedings of the Digital Forensics and Cyber Crime, Boston, MA, USA, 16–18 November 2022; Gladyshev, P., Goel, S., James, J., Markowsky, G., Johnson, D., Eds.; Springer: Cham, Switzerland, 2022; pp. 215–228.

42. Pang, C.; Zhang, T.; Yu, R.; Mao, B.; Xu, J. Ground Truth for Binary Disassembly is Not Easy. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; pp. 2479–2495.
43. Li, K.; Woo, M.; Jia, L. On the Generation of Disassembly Ground Truth and the Evaluation of Disassemblers. In Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation, Virtual Event, USA, 13 November 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 9–14, FEAST'20. [CrossRef]
44. Pei, K.; Guan, J.; Williams-King, D.; Yang, J.; Jana, S. XDA: Accurate, Robust Disassembly with Transfer Learning. In Proceedings of the Network and Distributed System Security Symposium (NDSS 21), Boston, MA, USA, 21–25 February 2021; pp. 1–18.
45. Cao, Y.; Liang, R.; Chen, K.; Hu, P. Boosting Neural Networks to Decompile Optimized Binaries. In Proceedings of the 38th Annual Computer Security Applications Conference, Austin, TX, USA, 5–9 December 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 508–518, ACSAC '22. [CrossRef]
46. Yu, S.; Qu, Y.; Hu, X.; Yin, H. DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; pp. 2709–2725.
47. Schloegel, M.; Blazytko, T.; Contag, M.; Aschermann, C.; Basler, J.; Holz, T.; Abbasi, A. Loki: Hardening Code Obfuscation Against Automated Attacks. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; pp. 3055–3073.
48. Xu, H.; Zhou, Y.; Ming, J.; Lyu, M. Layered obfuscation: A taxonomy of software obfuscation techniques for layered security. *Cybersecurity* **2020**, *3*, 9. [CrossRef]
49. Menguy, G.; Bardin, S.; Bonichon, R.; Lima, C.d.S. Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, 15–19 November 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 2513–2525, CCS '21. [CrossRef]
50. Yadegari, B.; Johannesmeyer, B.; Whitely, B.; Debray, S. A Generic Approach to Automatic Deobfuscation of Executable Code. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 674–691. [CrossRef]
51. Qian, Z.; Huang, K.; Wang, Q.F.; Zhang, X.Y. A survey of robust adversarial training in pattern recognition: Fundamental, theory, and methodologies. *Pattern Recognit.* **2022**, *131*, 108889. [CrossRef]
52. Bai, T.; Luo, J.; Zhao, J.; Wen, B.; Wang, Q. Recent Advances in Adversarial Training for Adversarial Robustness. In Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, Montreal, QC, Canada, 19–27 August 2021; pp. 4312–4321, IJCAI '21.
53. Luo, Z.; Wang, P.; Wang, B.; Tang, Y.; Xie, W.; Zhou, X. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search. In Proceedings of the Network and Distributed System Security (NDSS) Symposium, San Diego, CA, USA, 27 February–3 March 2023; pp. 1–18.
54. Xue, Y.; Xu, Z.; Chandramohan, M.; Liu, Y. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation. *IEEE Trans. Softw. Eng.* **2019**, *45*, 1125–1149. [CrossRef]
55. Wang, J.; Sharp, M.; Wu, C.; Zeng, Q.; Luo, L. Can a deep learning model for one architecture be used for others? retargeted-architecture binary code analysis. In Proceedings of the 32nd USENIX Conference on Security Symposium, Anaheim, CA, USA, 9–11 August 2023.