Towards Fine-Grained Compiler Identification with Neural Modeling

Borun Xie[1,2], Zhenzhou Tian[1,2*], Cong Gao[1,2], Lingwei Chen[3]
[1]School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an, China
[2]Shaanxi Key Laboratory of Network Data Analysis and Intelligent Processing, Xi'an, China
[3]College of Information Sciences and Technology, Pennsylvania State University, PA, USA
[*]Corresponding: tianzhenzhou@xupt.edu.cn

***Abstract*—Different compilers and optimization levels can be used to compile the source code. Revealed in reverse from the produced binaries, these compiler details facilitate essential binary analysis tasks, such as malware forensics and binary code similarity analysis. Most existing approaches adopt a signature matching based or machine learning based strategy to identify the compiler details, showing limits in either the detection accuracy or granularity. In this work, we propose NeuralCI (Neural modeling-based Compiler Identification) to perform the identification of compiler family and optimization level on individual functions. The basic idea is to formulate sophisticated neural networks to process abstracted instruction sequences generated using a lightweight function abstraction strategy. To evaluate the performance of NeuralCI, a large dataset consisting of 413,119 unique functions collected from real-world projects is constructed. The experiments show that NeuralCI achieves over 99% and 90% accuracy in identifying the compiler family and optimization level respectively, which outperforms most state-of-the-art function level compiler identification methods. Also, we explore for the first time the possibility of conducting compiler identification on binary code snippets rather than complete functions, where NeuralCI still achieves 96% accuracy, indicating its ability of capturing subtle yet significant features.**

Keywords-software forensics; binary code analysis; compiler identification; neural network

I. INTRODUCTION

In the software production process, diverse toolchains and toolchain settings can be adopted to transform the source code to the final binary. For example, different compilers like GCC and Clang as well as different compiler options like O0-O3 can be used by the developers. Besides, it is also a common practice to apply various kinds of code obfuscation techniques [1, 2] and packers [3, 4] in the binary production process.

Usually binaries produced with these different toolchains and toolchain settings exhibit significant differences when viewed in a straight way [5-7]. These differences just indicate that toolchain footprints are preserved during the source code to binary code translation process, enabling the possibility of revealing the toolchain and toolchain setting choices made during the production process of a binary. This task, which in the literature is called binary program provenance analysis, provides ways to spy on the specifics of the binary production process. A major subtask of it, compiler identification, which focuses on the compilation phase, attempts to infer from a piece of binary code the compiler-related details such as the specific compiler family, the optimization options etc.

Overall, relatively few works have been conducted on compiler identification, which mainly fall into two categories: signature matching based methods [8-10] and learning based methods [11-14]. The former, implemented in several reverse engineering tools like IDA [8] and PEiD [9], performs whole program level identification via exact matching of signatures that are manually constructed for certain compilers. Drawbacks of these kinds of methods lie in the stringent expertise in constructing a good enough compiler-specific signature as well as their coarse identification granularity. The latter formulates compiler identification as a machine learning task, which trains models to capture compiler-specific patterns, further with which to infer the compiler details on previously unseen binaries. For this kind of method, syntactic or structural features are extracted based on artificially defined templates such as idioms [11] which are short sequences of instructions with wildcards or graphlets [12] which are small subgraphs within the CFG (Control Flow Graph). The accuracy of these methods greatly depends on the quality of manually-crafted feature extraction strategies, where potential human-bias exists, resulting in capturing lots of irrelevant or redundant features for the compiler identification task meanwhile failing to capture closely relevant ones.

In recent years, tremendous successes have been witnessed of applying natural language processing techniques and deep learning models to various program analysis tasks [16-21]. In this paper, we attempt to adopt some of the most popular neural network structures to achieve fast and accurate fine-grained compiler identification on function level. Specifically, we feed typical Recurrent Neural Network (RNN) and Convolutional Neural Network (CNN) based structures with abstracted assembly instruction sequences to train classification models for inferring the compiler families and the optimization levels. Our intuition is based on the observation that co-occurring instructions together with their orderings that appear even in short instruction sequences form good enough signals of distinguishing different compliers or optimization levels, and here we resort to the neural models to capture them.

Our main contributions are summarized as following:

- We propose to reveal fine-grained compiler details for individual functions by designing a lightweight

DOI reference number: 10.18293/SEKE2020-141.

function abstraction strategy and adapting typical sequence-oriented neural networks. It alleviates the task complexity and human bias impacts by handing over the professional process of extracting and selecting features significant for compiler identification from the domain experts to the less human intervened neural networks.

- We have implemented the proposed methods as a tool called NeuralCI (**Neural** modeling based **C**ompiler **I**dentification), and evaluated its performance of revealing either the compiler family or the optimization level on a large dataset consisting of 413,119 unique functions that we constructed via processing a set of diverse real world projects. The experiments show that NeuralCI outperforms most state-of-the-art function level compiler identification methods. It achieves above 99% and 90% accuracy in identifying compiler family and optimization level respectively.
- We explore for the first time the possibility of conducting compiler identification on arbitrary binary code snippets rather than complete functions. It shows that NeuralCI achieves 96% accuracy, indicating that it can capture very subtle yet significant features.

The remainder of this paper is organized as follows: Section II summarizes the related works. Section III describes in detail our proposed approach. The experimental evaluation conducted are presented in Section IV. Finally, we conclude the paper in Section V.

II. Related Work

In general, existing works on compiler identification can be divided into two classes: signature matching based methods and learning based methods.

A. Signature Matching Based Methods

The signature matching based methods [8-10] search the binary program against a corpus of manually constructed signatures for exact matching, and attribute to the whole program the compiler label corresponding to the matched signature string. This kind of method has been implemented in several reverse engineering tools, such as IDA Pro [8], PEiD [9] and LANGUAGE 2000 [10], in consideration of its high detection efficiency and low cost. Drawbacks of these methods lie in the stringent expertise and labor work in constructing a good enough compiler-specific signature, as well as the easily affected accuracy due to slight differences between signatures. Besides, the signatures usually depend on the metadata or details of program headers, which can be easily altered or become unavailable in stripped binaries. Moreover, these tools identify compilers on the whole binary, while a program can be produced with multiple compilers in scenarios such as statically linking library code to produce the final binary program.

B. Learning Based Methods

This type of method formulates compiler identification as a machine learning task performed on (in most instances stripped) binaries, based on the belief that the resulting binaries implicit features reflecting design and implementation decisions of the certain compiler which are used to produce the binaries. Specifically, they train models that capture compiler-specific patterns, further with which to infer the compiler provenance on previously unseen binaries.

The pioneering work [11] adopting this type of approach was conducted by Rosenblum et al. that manually defined a set of idioms (short sequences of instructions with wildcards) and utilized mutual information calculation to capture and select significant patterns indicative of the source compiler of the program binaries. High accuracy is observed for inferring the compiler families, but we have no idea of its performance on optimization levels identification as no evaluation was conducted. ORIGIN [12] achieved superior accuracy in recovering the compiler details by introducing graphlets (small and non-isomorphic subgraphs within the CFG) in addition to idioms so as to capture additional structural features. Hidden Markov models were learnt via observing the differences in the type and frequency of instructions comprising the binaries compiled with different compilers, and proved to be accurate in identifying the compiler family for a whole program [14, 15]. However, for each individual compiler family a corresponding separate model needs to be learnt. Also, these models do not extract information regarding the optimization levels. To improve efficiency in terms of computational resources and detection time, BinComp [13] adopted a stratified approach to infer different compiler details on different granularity. It identifies compiler family for the whole program via exact matching of signatures, and conducts compiler version and optimization level detection for compiler-related functions. However, the compiler-related functions constitute only a very small portion of all functions, making it largely impractical in handling real world programs where user defined functions hold the principal status. Basically, accuracy of these methods greatly depends on the quality of manually-crafted feature extraction strategies, where potential human-bias exists, resulting in capturing lots of irrelevant or redundant features for the compiler provenance task meanwhile failing to capture closely relevant ones.

In recent years, significant successes have been witnessed of applying deep learning techniques to the domain of binary program analysis [16-21]. BinEye [16] is one of the few works that utilize neural models to achieve compiler identification. It combines word embedding and position embedding to encode the raw bytes of an object file, and then utilizes CNN to learn a model that supports optimization level recognition on each individual object file. Our work differs in that we achieve finer grained identification of both the compiler family and the optimization level for each individual function by adopting an abstraction strategy that operates on assembly instructions rather than the raw bytes. Structure2Vec [19] utilizes a graph embedding network to transform the function CFGs into vectors, which are then fed into a dense layer to train a classifier for compiler family identification. Compared to this work, we operate directly on the instructions comprising a function with a lightweight abstraction strategy, and adopt the much faster sequence-oriented neural networks to train models for identifying the optimization level besides just the compiler family as did by Structure2Vec.

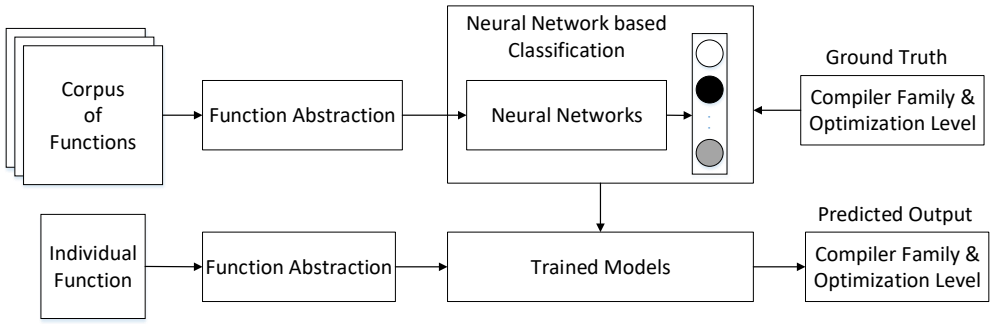

Figure 1. The basic framework of NeuralCI

III. THE APPROACH

Figure 1 depicts the overall architecture of NeuralCI, which consists of two phases: the training phase as illustrated in the top half subfigure, and the detection phase as illustrated in the half bottom subfigure. The training phase consists of three steps. As a deep learning-based method, the first step is to construct a high-quality database comprised of labeled functions which shall be discussed in detail in Section IV.A. The second step takes as input each raw function and outputs an abstracted instruction sequence via a light-weight abstraction strategy implemented in the function abstraction module. Then these abstracted sequences together with their ground truth labels are fed into the neural network based classification module to train compiler identification models. The detection phase is much simpler, which takes in an individual function, processes it with the function abstraction and utilizes the trained model to produce a predicted output. In the following, we discuss the details of the function abstraction module and neural network based classification module respectively.

A. Function Abstraction

A function must be represented in certain forms such that it can be processed for further analysis. The typical ways include using the raw byte sequence, the assembly instruction sequence or the control flow graph [19] to depict a function. According to the findings in [15], different compilers tend to use distinguishable assembly instructions. For example, *call* instruction occurs frequently in GCC-generated assembly code, while the Clang assembly uses *callq* instead. Also, as indicated by the pretty good compiler family identification accuracy in [12], short assembly instruction sequences successfully capture compiler-related features. Thus, in this work we choose to use the assembly instruction sequence as the representation of each function, and we use IDA Pro for the parsing [1]. That is, a function f will be represented as $f = \{ins_1, ins_2, \cdots, ins_n\}$, where n denotes the number of instructions within the function, and each instruction ins_i consists of an opcode (i.e. mnemonic) and a list of operands.

[1] We assume a reliable way of identifying the function boundaries, the instruction boundaries, as well as the correct parsing of each instruction, by using the best commercial reverse engineering tool IDA Pro. The correct disassembly of binaries is still a complex and open problem, but are beyond the scope of this paper.

However, as has been confirmed in many existing binary analysis tasks [19, 20], it is usually not wise to work directly on the raw assembly instructions. For our case, we want to capture features reflecting the compiler details rather than the functionality of the function. That is, we do not care whether a value 6 is assigned to register *eax* or a value 10 is assigned. So the two instructions "*mov eax, 6*" and "*mov eax, 10*" should be considered the same. Besides, the memory addresses (e.g. the target of *jmp* instructions) are meaningless but just noises that bring adverse impacts. Meanwhile, to prevent introducing too much human bias, we choose to process the raw instruction sequence with a light-weight abstraction strategy.

Specifically, we do abstraction to each assembly instruction in a function with the following rules:

- The mnemonics remain unchanged.
- All registers in the operands remain unchanged.
- All base memory addresses in the operands are substituted with the symbol *MEM*.
- All immediate values in the operands are substituted with the symbol *IMM*.

As an example, with the above abstraction rules, the instruction "*add eax, 6*" will become "*add eax, IMM*", the instruction "*mov ebx, [0x3435422]*" will become "*mov ebx, MEM*", while the instruction "*mov eax, [ebp-20]*" will become "*mov eax, [ebp-IMM]*".

B. Neural Models for Compiler Identification

Given a set of abstracted assembly instruction sequences, it is promising to utilize skip-gram [22] to learn the embedding for each instruction, explore max-pooling, averaging or concatenation to aggregate the embeddings for each sequence, and then feed them to any classification model for compiler identification. However, it still faces the following two limitations: (1) skip-gram assigns each instruction a static embedding vector, which is not context-aware to different sequences it interacts with; this may fail to learn the compiler-related features; (2) since instruction sequences are abstracted from functions, they may not only enjoy local instruction associations and correlations, but also global or long-range instruction dependency; in this respect, it calls for sequence learning models to better capture the representative compiler-specific patterns and features from instruction sequences for compiler identification. As advanced neural network structures,

both RNN and CNN have achieved great success in sequence learning. As such, in this work, we design an RNN model and a CNN model respectively to learn the semantic and structural information of instruction sequences and thus leverage these advances to identify their compilers.

1) RNN Model: RNN is known to learn the sequential dependency, and strict to align the positions and contexts for the instances in the input sequences. Considering that some instructions may play more significant roles in the function or some instructions may be uniquely generated by specific compilers, RNN is able to attend such instructions and learn a comprehensive and contextualized embedding for the whole instruction sequence. In this work, we employ Long Short-term Memory (LSTM) or Gated Recurrent Unit (GRU), either of which is an architecture designed for RNN to address the vanishing and exploding gradient issue. The structure of our RNN model is shown in Figure 2. Each instruction in the input sequence is first embedded in vector space. Afterwards, the model reads the input instruction sequence through LSTM/GRU units to obtain the summary vector, which is then fed to a Softmax layer to predict the real compiler. The training loss is adopted to measure the correctness of sequence learning and compiler prediction. During the training process, dropout is also applied to prevent the neural network from overfitting.

2) CNN Model: Different from RNN, CNN is known to learn the local correlations with shared weights and utilize pooling mechanism to greatly reduce the number of parameters needed to find important local patterns. In other words, CNN is able to attend those frequently co-occurring instructions in the short sequences. In our model formulation, we further take advantage of different kernel-size filters to thoroughly extract interacted salient features among different instruction grams to capture the behaviors of compilers. The structure of our CNN model is shown in Figure 3. Each instruction sequence is first transformed into a matrix, where each row of the matrix is the instruction's embedding. We take such an embedding matrix as input fed to the CNN architecture to learn the higher-level concept. In the convolutional layer, the raw instruction feature matrix gets convoluted by different kernels of size 2, 3 and 4 such that different views of feature patterns can be extracted in parallel, which are then passed through 1-maxpooling layers for dimensionality reduction. The resulting representations are concatenated through a dense layer to be fed to a Softmax layer for compiler prediction. The CNN model is trained using the instruction sequences with ground truth.

IV. Experiments and Evaluation

A. Dataset Construction

To evaluate the performance of NeuralCI, we collected 9 widely used C/C++ open source projects, including coreutils 8.31, curl 7.65.3, FreeImage 3.17, git 2.22, libpng 1.6.3, pigz 2.4, x264, vim 8.1.19 and sqlite 3.22, as the basics to construct the dataset. To be specific, we process these projects with the following steps:

- Two different compilers involving multiple versions including GCC (4.6.3, 4.7.4, 4.8.5, 4.9.4, 5.5.0, 6.5.0, 7.4.0) and Clang (3.8, 5.0, 6.0), as well as varying compiler optimization levels (O0, O1, O2, O3) are used as the toolchain settings to compile each project.
- IDA Pro is then used to identify and extract functions from each binary. Also, we get rid of trivial functions (functions containing just a few instructions, such as the stub functions) that are meaningless to analyze. We consider functions containing less than 10 instructions as trivial in our current setting.
- To avoid the neural models see during the training phase functions that are really similar to the ones in the testing phase, which if not properly accounted for can inflate the performance metrics, we only keep unique functions. Specifically, a function is considered redundant if it has the same abstracted instructions as any other functions'. Then we label each remaining function with the compiler settings used to compile the binary that the function resides in.

With these settings, we finally construct a dataset comprised of totally 413,119 unique functions.

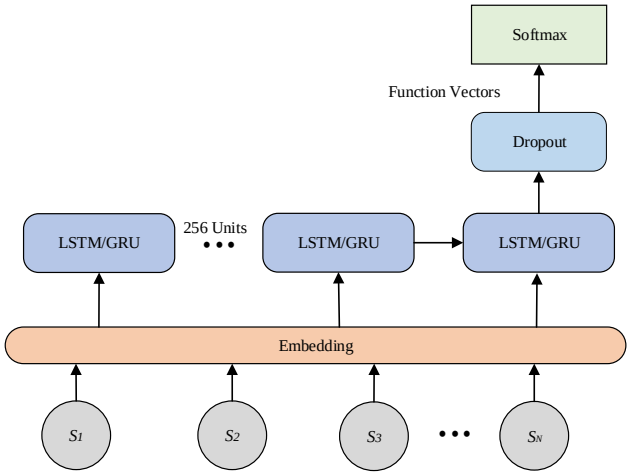

Figure 2. RNN-based model

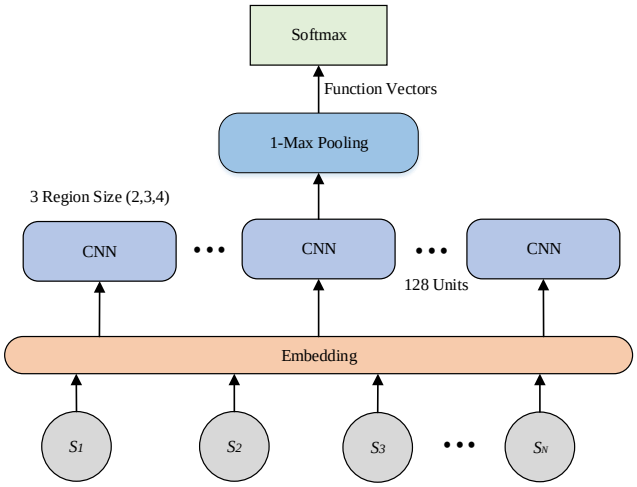

Figure 3. CNN-based model

B. Implementation Details and Experimental Settings

We have implemented NeuralCI as a prototype tool. It utilizes IDA Pro for the parsing of binaries to obtain functions as well as the raw assembly instructions. The function abstraction module is implemented in Java, and the neural modeling module is implemented using Python and the Tensorflow framework. Skip-gram model provided by gensim is used to generate the instruction embedding vectors with the embedding size setting to 100.

For the experimental settings, we randomly split the whole dataset into training set, validation set and testing set according to a percentage of 70%, 15% and 15% respectively. The neural models are trained with a RTX2080Ti GPU card using a batch size of 500, learning rate 0.001 and Adam optimizer for 100 epochs (Note that for each epoch the training samples are shuffled and accuracy on the validation set is calculated). Then we take the model with the best validation accuracy as the final model to be further evaluated on the testing set with respect to performance metrics including accuracy, precision, recall and f1-score.

C. Evaluation

In the following parts, firstly we evaluate the performance of NeuralCI in identifying the compiler family and optimization level respectively and get it compared against state-of-the-art function level compiler identification methods. Then we further explore for the first time the applicability of NeuralCI on compiler identification of arbitrary binary code snippets which can be just a part of a complete function. Considering that several different neural structures are implemented in NeuralCI, we use NeuralCI_x where *x* can be LSTM, GRU or CNN to get them distinguished.

1)Performance of Identifying Compiler Family: In this experiment, we take the compiler family that each function is compiled with as the ground-truth, and get the NeuralCI models trained and evaluated by adopting the experimental settings as described in Section IV.B. Also, we compare them against existing methods including Structure2Vec, Idioms, Graphlets and BinComp that support function level compiler family identfication. Table I summarizes the experimental results[2]. As it shows, the three NeuralCI models achieve nearly perfect accuracy and f1-score in identifying the compiler family and they all outperform existing methods in terms of detection accuracy. Also, the CNN based one $\text{Neural}_{\text{CNN}}$ exhibits the best performance with an accuracy of 99.5% and f1-score of 0.995, an improvement of 6.3% against Idioms and the least improvement of 0.7% against Graphlets.

2)Performance of Identifying Optimization Level: In this experiment, the optimization levels of certain compilers are taken as the ground-truth to train and evaluate NeuralCI. We do not use the default 4-level optimization option setting, but adopt the same strategy as in works [12, 13] that condense the 4 optimization levels to 2 classes 'low' and 'high', considering the findings presented in existing studies [12, 13] that it is difficult to distingush between O2 and O3 compiled binaries. That is, O0 and O1 will be considered as the low optimization class, while O2 and O3 will be considered as the high optimization class. Similarly, we compare NeuralCI with methods having function-level compiler optmization level identification results. As summarized in Table II the detection results on GCC optimzation levels, relatively good detection accuracy of above 90% are observed for the NeuralCI models, while the Graphlets method achieves the best accuracy of 97.1% according to their original evaluation result. But as the reproduction evaluation conducted by BinComp on a different dataset shows, the Graphlets method achieves an f-score of just 0.62, much lower than those of NeuralCI's. Similar results can be observed for evaluation on the Clang optimization levels in Table III, where an accuracy of above 90% is achieved for the NeuralCI models. There's no data for other methods, as they didn't evaluate on the Clang compiler.

TABLE I
COMPILER FAMILY IDENTIFICATION RESULTS

Model	Accuracy	Precision	Recall	F1-Score
$\text{NeuralCI}_{\text{LSTM}}$	99.4%	0.994	0.993	0.994
$\text{NeuralCI}_{\text{GRU}}$	99.3%	0.994	0.99	0.994
$\text{NeuralCI}_{\text{CNN}}$	**99.5%**	**0.995**	**0.995**	**0.995**
Structure2Vec	98.2%	0.98	0.98	0.98
Idioms	93.2%	-	-	-
Graphlets	98.7%	-	-	-
BinComp	97.0%	-	-	-

TABLE II
OPTIMIZATION LEVEL IDENTIFICATION RESULTS FOR GCC

Model	Accuracy	Precision	Recall	F1-Score
$\text{NeuralCI}_{\text{LSTM}}$	91.3%	**0.914**	0.912	**0.914**
$\text{NeuralCI}_{\text{GRU}}$	91.2%	0.913	**0.916**	**0.914**
$\text{NeuralCI}_{\text{CNN}}$	90.9%	**0.914**	0.907	0.910
Graphlets	**97.1%**	-	-	-
BinComp	91.0%	-	-	-

TABLE III
OPTIMIZATION LEVEL IDENTIFICATION RESULTS FOR CLANG

Model	Accuracy	Precision	Recall	F1-Score
$\text{NeuralCI}_{\text{LSTM}}$	**0.914**	0.920	**0.917**	0.908
$\text{NeuralCI}_{\text{GRU}}$	0.903	**0.925**	0.912	**0.924**
$\text{NeuralCI}_{\text{CNN}}$	0.900	0.915	0.905	0.903

3)Evaluation on Isolated Binary Code Snippet: The prologue (the preparation of stacks and registers to be used)

[2] It should be noted that most existing methods do not provide a public access to their source code implementation or the dataset they used. So we give the best performance exhibited in their original evaluations rather than conducting a direct comparison with them. Also, not all performance metrics as we evaluated against are used in their original work, we mark these missing data with a symbol '-' in Table I as well as the following tables.

and epilogue (the lines of code appearing at the end of a function for restoring the stack and registers to the state before the function is called) of a function play important roles in many binary analysis tasks. Also, as shown by the analysis presented in the works of Austin[15] and Toderici [16], *push*, *mov* and *pop* instructions (which are the main components of prologues and epilogues) show significant effect in distingushing different compilers. So in this experiment we explore whether NeuralCI still works on arbitrary binary code snippets that may not contain function prologue and epilogue. To achieve that, we apply NeuralCI to identify compiler family for each basic block containing no less than 10 instructions. As depicted in Table IV, reduced performance is observed compared with the results on individual functions, but still pretty good accuracy (around 96%) and f1-score (around 0.96) are achieved. It indicates that NeuralCI can capture very subtle yet significant features which may otherwise be missed by artifically crafted feature extraction and selection strategies.

TABLE IV
COMPILER FAMILY IDENTIFICAITON ON INDIVIDUAL BASIC BLOCKS

Model	Accuracy	Precision	Recall	F1-Score
$\text{NeuralCI}_{\text{LSTM}}$	**0.962**	**0.962**	**0.961**	**0.961**
$\text{NeuralCI}_{\text{GRU}}$	0.960	0.959	0.959	0.960
$\text{NeuralCI}_{\text{CNN}}$	0.961	0.960	0.960	**0.961**

V. CONCLUSION

In this work, we attempt to solve the problem of fine-grained compiler identification by feeding in typical neural networks with abstracted instruction sequences generated with a light-weight function abstraction strategy. We implement our methods in a prototype tool NeuralCI, and get its performance evaluated on a large dataset consisting of totally 413,119 unique functions. As the experimental evaluation shows, NeuralCI outperforms most state-of-the-art function level compiler identification methods. It achieves above 99% and 90% accuracy in identifying compiler family and optimization level respectively. Moreover, the evaluation we conducted of applying NeuralCI for the tougher task that infers compiler family from arbitrary binary code snippet achieves rather good accuracy of 96% and f1-score of 0.96. Future works will focus on the model interpretability as well as designing more powerful neural nets such as network structures with attention mechanism.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China (61702414), the Natural Science Basic Research Program of Shaanxi (2018JQ6078, 2020GY-010), the Science and Technology of Xi'an (2019218114GXRC017CG018-GXYD17.16), the International Science and Technology Cooperation Program of Shaanxi (2018KW-049, 2019KW-008), and Key Research and Development Program of Shaanxi (2019ZDLGY07-08).

REFERENCES

[1] Schrittwieser S, Katzenbeisser S, Kinder J, Merzdovnik G, Weippl, E. Protecting software through obfuscation: Can it keep pace with progress in code analysis?[J]. ACM Computing Surveys (CSUR), 2016, 49: 1-37.

[2] Schrittwieser S, Katzenbeisser S. Code obfuscation against static and dynamic reverse engineering[C]//International workshop on information hiding. Springer, Berlin, Heidelberg, 2011: 270-284.

[3] Roundy K A, Miller B P. Binary-code obfuscations in prevalent packer tools[J]. ACM Computing Surveys (CSUR), 2013, 46(1): 1-32.

[4] Ugarte-Pedrero, X, Balzarotti, D, Santos, I, & Bringas, P. G. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers[C]//2015 IEEE Symposium on Security and Privacy. IEEE, 2015: 659-673.

[5] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang. Neural machine translation inspired binary code similarity comparison beyond functions pairs[J].arXiv preprint arXiv:1808.04706,2018.

[6] Z. Z. Tian, T. Liu, Q. H. Zheng, E. Zhuang, M. Fan, and Z. J. Yang. Reviving sequential program birthmarking for multithreaded software plagiarism detection[J]. IEEE Transactions on Software Engineering, 2017, 44(5): 491-511.

[7] Z. Z. Tian, Q. H. Zheng, T. Liu, M. Fan, E. Y. Zhuang, and Z. J. Yang, Software plagiarism detection with birthmarks based on dynamic key instruction sequences[J]. IEEE Transactions on Software Engineering, 2015, 41(12): 1217-1235.

[8] IDA. Available: https://www.hexrays.com/products/ida/index.shtml.

[9] PEiD. Available: https://www.aldeid.com/wiki/PEiD.

[10] LANGUAGE 2000. Available: https://farrokhi.net/language/.

[11] Rosenblum N E, Miller B P, Zhu X. Extracting compiler provenance from program binaries[C]//Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. 2010: 21-28.

[12] Rosenblum N, Miller B P, Zhu X. Recovering the toolchain provenance of binary code[C]//Proceedings of the 2011 International Symposium on Software Testing and Analysis. 2011: 100-110.

[13] Rahimian A, Shirani P, Alrbaee S, Wang L, & Debbabi M. Bincomp: A stratified approach to compiler provenance attribution[J]. Digital Investigation, 2015, 14: S146-S155.

[14] Toderici A H, Stamp M. Chi-squared distance and metamorphic virus detection[J]. Journal of Computer Virology and Hacking Techniques, 2013, 9(1): 1-14.

[15] Austin T H, Filiol E, Josse S, et al. Exploring hidden markov models for virus analysis: a semantic approach[C]//2013 46th Hawaii International Conference on System Sciences. IEEE, 2013: 5039-5048.

[16] Yang S, Shi Z, Zhang G, et al. Understand Code Style: Efficient CNN-Based Compiler Optimization Recognition System[C]//2019 IEEE International Conference on Communications (ICC). IEEE, 2019: 1-6.

[17] Shin E C R, Song D, Moazzezi R. Recognizing functions in binaries with neural networks[C]//24th USENIX Security Symposium (USENIX Security 15). 2015: 611-626.

[18] Chua Z L, Shen S, Saxena P, & Liang Z. Neural nets can learn function type signatures from binaries[C]//26th USENIX Security Symposium (USENIX Security 17). 2017: 99-116.

[19] Massarelli L, Di Luna G A, Petroni F, et al. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis[C]//Proceedings of the 2nd Workshop on Binary Analysis Research (BAR). 2019.

[20] Zuo F, Li X, Young P, et al. Neural machine translation inspired binary code similarity comparison beyond function pairs[C]//Proceedings of the Network and Distributed Systems Security Symposium (NDSS), 2019.

[21] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 141-151: ACM.

[22] Mikolov T, Chen K, Corrado G, et al. Efficient estimation of word representations in vector space[J]. arXiv preprint arXiv:1301.3781, 2013.