IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 11, 2021, accepted March 24, 2021, date of publication March 29, 2021, date of current version April 5, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3069227

Fine-Grained Compiler Identification With Sequence-Oriented Neural Modeling

ZHENZHOU TIAN[1,2], YAQIAN HUANG[1,2], BORUN XIE[1,2], YANPING CHEN[1,2], LINGWEI CHEN[3], AND DINGHAO WU[3], (Member, IEEE)

[1]School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an 710121, China
[2]Shaanxi Key Laboratory of Network Data Analysis and Intelligent Processing, Xi'an 710121, China
[3]College of Information Sciences and Technology, The Pennsylvania State University, University Park, PA 16802, USA

Corresponding author: Zhenzhou Tian (tianzhenzhou@xupt.edu.cn)

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

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 analysis and software forensics. 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 infer these compiler details including compiler family, optimization level and compiler version on individual functions. The basic idea is to formulate sequence-oriented neural networks to process normalized instruction sequences generated using a lightweight function abstraction strategy. To evaluate the performance of NeuralCI, a large dataset consisting of 854,858 unique functions collected from 19 widely used real-world projects is constructed. The experiments show that NeuralCI achieves averagely 98.6% accuracy in identifying the compiler family, 95.3% accuracy in identifying the optimization level, 88.7% accuracy in identifying the compiler version, 94.8% accuracy in identifying the compiler family and optimization level, and 83.0% accuracy in identifying all compiler components simultaneously, outperforming existing function level compiler identification methods in terms of both detection accuracy and comprehensiveness.

INDEX TERMS 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, in consideration of the stability and performance issue, the size requirement on the produced binary and their familiarity with the tools. Besides, it is also a common practice to apply various kinds of code obfuscation techniques [31], [32] and packers [30], [41] in the binary production process, with which developers protect the core algorithms and codes from being easily reverse engineered [13], [43], while malware writers use them to hide malicious part in the binary from being easily detected by security analysis tools [20], [33].

Usually, binaries produced with these different toolchains and toolchain settings exhibit significant differences when viewed in a straight way [37], [38], [51]. These differences just indicate that toolchain footprints are preserved during the translation process from source code to binary code, 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. As its major subtask to focus on the compilation phase, compiler identification attempts to infer from a piece of binary code the compiler-related details such as the specific compiler family, the optimization options, etc., which can facilitate essential binary analysis tasks such

The associate editor coordinating the review of this manuscript and approving it for publication was Claudia Raibulet.

as malware analysis and software forensics. For example, in the scenario of software plagiarism detection, where the source code of the plaintiff program is generally accessible, we can eliminate the interference of different compiler settings to the similarity analysis by recompiling the source code with the same compiler settings identified from the defendant binary program.

Overall, there have been relatively few works conducted on compiler identification, which mainly fall into two categories: signature matching based methods [6]–[8] and learning based methods [27]–[29], [39]. The former, implemented in several reverse engineering tools like IDA [6] and PEiD [8], performs whole program level identification via matching against a corpus of generic and rigid signatures. The drawbacks to these kinds of methods are 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, and infer the compiler details on previously unseen binaries. For such kinds of methods, syntactic or structural features are extracted based on artificially defined templates, such as idioms [29] which are short sequences of instructions with wildcards, or graphlets [28] which are small subgraphs within the CFG (Control Flow Graph). As typical feature engineering based methods, their effectiveness, to a large extent, depends on the quality of expert-defined feature extraction templates, where more domain-specific knowledge is required.

In recent years, tremendous successes have been witnessed of applying natural language processing techniques and deep learning models to various program analysis tasks [16], [23], [34], [45], [48], which leverage many layers of non-linearities to capture invariances from transformation in the raw input space, and have thus automatically boosted the semantic richness for the learned representations. Inspired by these great successes, 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 convolutional neural network (CNN) and recurrent neural network (RNN) based structures with normalized assembly instruction sequences to train classification models for inferring the compiler families, the optimization levels, and the compiler versions. Our intuition is based on the observation that co-occurring instructions together with their orderings in short instruction sequences form good enough signals of distinguishing different compilers or optimization levels, which can be substantially captured by neural models. Our main contributions are summarized as following:

- We propose to reveal fine-grained compiler details for individual functions by designing a lightweight function abstraction strategy and leveraging 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 elaborate two neural network structures CNN and RNN to solve compiler identification problem. The former contains three variations (i.e., one naïve and two attention augmented ones), and the latter contains four variations (i.e., one naïve and three attention augmented ones). All of them are implemented in a tool called (**Neural** modeling based **C**ompiler **I**dentification).
- We construct a large dataset consisting of 854,858 unique functions by processing a set of diverse real world projects, and systematically evaluate and compare the performance of the proposed methods with respect to revealing compiler family, optimization level, compiler version, and compiler setting combination respectively. The experimental evaluation shows that NeuralCI achieves promising performance of revealing these fine-grained compiler details and outperforms existing function level compiler identification methods in both detection performance and comprehensiveness. Our dataset and source code implementation of NeuralCI have been made public at https://github.com/zztian007/NeuralCI to facilitate further researches.

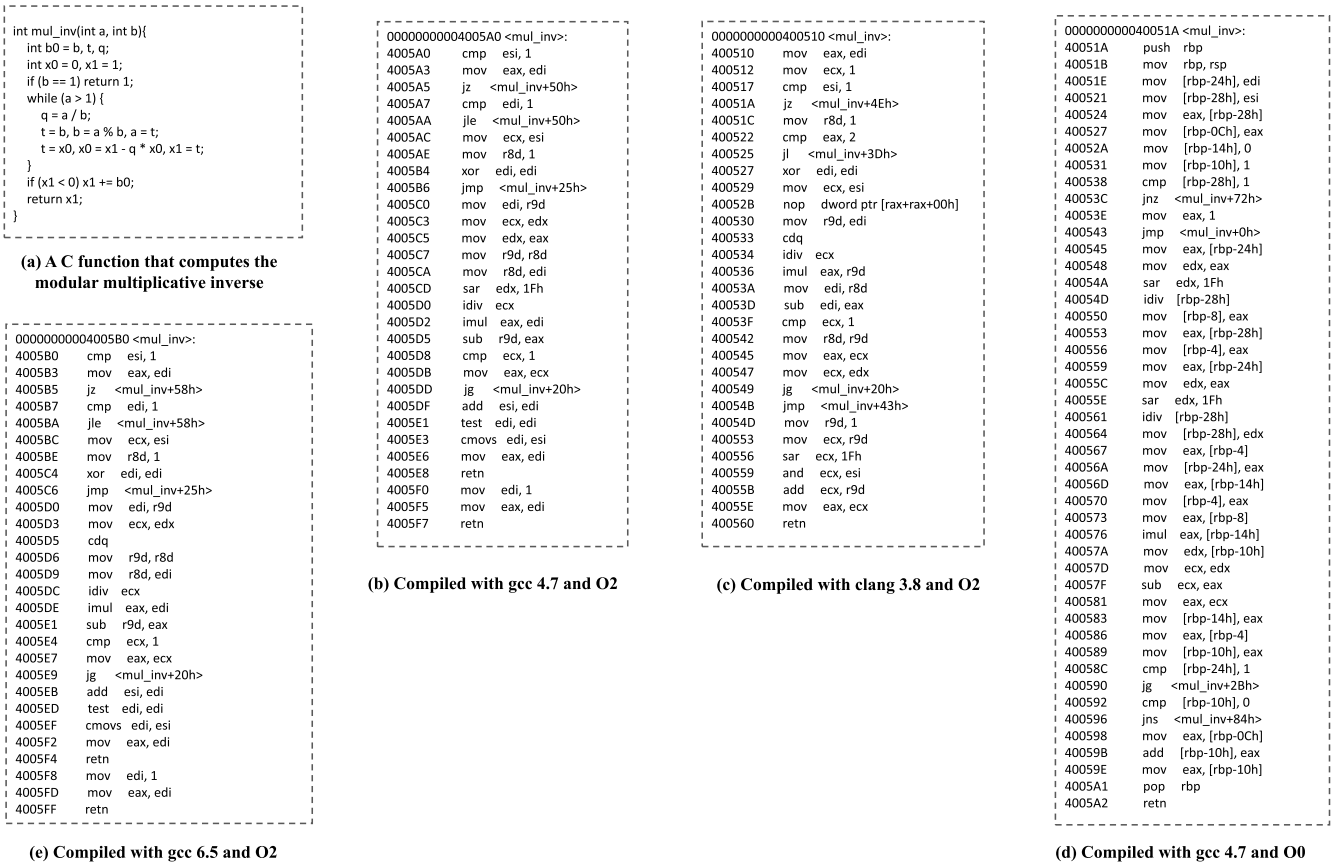
The rest of this paper is organized as follows: Section II formulates the problem to solve and our design overview. Section III and Section IV describe in detail the considerations and designs of the function abstraction and the neural network models respectively. The dataset construction and experimental evaluations are presented in Section V. Section VI mainly discusses the limitations of our method. Section VII summarizes the related works. Finally, Section VIII concludes.

II. PROBLEM DEFINITION AND DESIGN OVERVIEW

A. PROBLEM OVERVIEW

The goal of compiler identification is to reveal in reverse from the final produced binary the compiler-related details applied in processing the program source code. The feasibility of this task lies in the usually significant differences imposed by different compiler and optimization settings. Such differences across homogenous binaries, which drive and meanwhile bother massive binary code similarity analysis researches, just indicate that the different mechanisms, heuristics and design choices within certain compilers will ultimately manifest on their outputs, providing evidence of program provenance.

To give a more intuitive sense of the differences, Figure 1 lists the corresponding binary codes produced with several different compiler and optimization settings from a simple C function. Take the binary code in Figure 1(b) (compiled with gcc 4.7 and the generally recommended optimization level O2) as the base, we compare it against the binary code generated with every other compiler setting using the binary code comparison tool Bindiff. The detected similarity scores



```
int mul_inv(int a, int b){
    int b0 = b, t, q;
    int x0 = 0, x1 = 1;
    if (b == 1) return 1;
    while (a > 1) {
        q = a / b;
        t = b, b = a % b, a = t;
        t = x0, x0 = x1 - q * x0, x1 = t;
    }
    if (x1 < 0) x1 += b0;
    return x1;
}
```

(a) A C function that computes the modular multiplicative inverse

```
00000000004005A0 <mul_inv>:
4005A0      cmp    esi, 1
4005A3      mov    eax, edi
4005A5      jz     <mul_inv+50h>
4005A7      cmp    edi, 1
4005AA      jle    <mul_inv+50h>
4005AC      mov    ecx, esi
4005AE      mov    r8d, 1
4005B4      xor    edi, edi
4005B6      jmp    <mul_inv+25h>
4005C0      mov    edi, r9d
4005C3      mov    ecx, edx
4005C5      mov    edx, eax
4005C7      mov    r9d, r8d
4005CA      mov    r8d, edi
4005CD      sar    edx, 1Fh
4005D0      idiv   ecx
4005D2      imul   eax, edi
4005D5      sub    r9d, eax
4005D8      cmp    ecx, 1
4005DB      mov    eax, ecx
4005DD      jg     <mul_inv+20h>
4005DF      add    esi, edi
4005E1      test   edi, edi
4005E3      cmovs  edi, esi
4005E6      mov    eax, edi
4005E8      retn
4005F0      mov    edi, 1
4005F5      mov    eax, edi
4005F7      retn
```

(b) Compiled with gcc 4.7 and O2

```
0000000000400510 <mul_inv>:
400510      mov    eax, edi
400512      mov    ecx, 1
400517      cmp    esi, 1
40051A      jz     <mul_inv+4Eh>
40051C      mov    r8d, 1
400522      cmp    eax, 2
400525      jl     <mul_inv+3Dh>
400527      xor    edi, edi
400529      mov    ecx, esi
40052B      nop    dword ptr [rax+rax+00h]
400530      mov    r9d, edi
400533      cdq
400534      idiv   ecx
400536      imul   eax, r9d
40053A      mov    edi, r8d
40053D      sub    edi, eax
40053F      cmp    ecx, 1
400542      mov    r8d, r9d
400545      mov    eax, ecx
400547      mov    ecx, edx
400549      jg     <mul_inv+20h>
40054B      jmp    <mul_inv+43h>
40054D      mov    r9d, 1
400553      mov    ecx, r9d
400556      sar    ecx, 1Fh
400559      and    ecx, esi
40055B      add    ecx, r9d
40055E      mov    eax, ecx
400560      retn
```

(c) Compiled with clang 3.8 and O2

```
00000000004005B0 <mul_inv>:
4005B0      cmp    esi, 1
4005B3      mov    eax, edi
4005B5      jz     <mul_inv+58h>
4005B7      cmp    edi, 1
4005BA      jle    <mul_inv+58h>
4005BC      mov    ecx, esi
4005BE      mov    r8d, 1
4005C4      xor    edi, edi
4005C6      jmp    <mul_inv+25h>
4005D0      mov    edi, r9d
4005D3      mov    ecx, edx
4005D5      cdq
4005D6      mov    r9d, r8d
4005D9      mov    r8d, edi
4005DC      idiv   ecx
4005DE      imul   eax, edi
4005E1      sub    r9d, eax
4005E4      cmp    ecx, 1
4005E7      mov    eax, ecx
4005E9      jg     <mul_inv+20h>
4005EB      add    esi, edi
4005ED      test   edi, edi
4005EF      cmovs  edi, esi
4005F2      mov    eax, edi
4005F4      retn
4005F8      mov    edi, 1
4005FD      mov    eax, edi
4005FF      retn
```

(e) Compiled with gcc 6.5 and O2

```
000000000040051A <mul_inv>:
40051A      push   rbp
40051B      mov    rbp, rsp
40051E      mov    [rbp-24h], edi
400521      mov    [rbp-28h], esi
400524      mov    eax, [rbp-28h]
400527      mov    [rbp-0Ch], eax
40052A      mov    [rbp-14h], 0
400531      mov    [rbp-10h], 1
400538      cmp    [rbp-28h], 1
40053C      jnz    <mul_inv+72h>
40053E      mov    eax, 1
400543      jmp    <mul_inv+0h>
400545      mov    eax, [rbp-24h]
400548      mov    edx, eax
40054A      sar    edx, 1Fh
40054D      idiv   [rbp-28h]
400550      mov    [rbp-8], eax
400553      mov    eax, [rbp-28h]
400556      mov    [rbp-4], eax
400559      mov    eax, [rbp-24h]
40055C      mov    edx, eax
40055E      sar    edx, 1Fh
400561      idiv   [rbp-28h]
400564      mov    [rbp-28h], edx
400567      mov    eax, [rbp-4]
40056A      mov    [rbp-24h], eax
40056D      mov    eax, [rbp-14h]
400570      mov    [rbp-4], eax
400573      mov    eax, [rbp-8]
400576      imul   eax, [rbp-14h]
40057A      mov    edx, [rbp-10h]
40057D      mov    ecx, edx
40057F      sub    ecx, eax
400581      mov    eax, ecx
400583      mov    [rbp-14h], eax
400586      mov    eax, [rbp-4]
400589      mov    [rbp-10h], eax
40058C      cmp    [rbp-24h], 1
400590      jg     <mul_inv+2Bh>
400592      cmp    [rbp-10h], 0
400596      jns    <mul_inv+84h>
400598      mov    eax, [rbp-0Ch]
40059B      add    [rbp-10h], eax
40059E      mov    eax, [rbp-10h]
4005A1      pop    rbp
4005A2      retn
```

(d) Compiled with gcc 4.7 and O0

FIGURE 1. An example of C function from http://rosettacode.org/wiki/Modular_inverse#C, and its corresponding assembly codes under different compiler settings.

are 0.76, 0.23, and 0.98 respectively for the binary code pairs ⟨1b, 1c⟩, ⟨1b, 1d⟩ and ⟨1b, 1e⟩ (we use ⟨1b, 1c⟩ to denote the pair of binary codes shown in Figure 1(b) and Figure 1(c) for the sake of simplicity), illustrating notable yet varying impacts on the produced binaries via changing the compiler family, compiler optimization level or even the compiler version.

More specifically, it can be observed that the compiled binaries are distinguishing in the length of their function bodies, the specific instructions and instruction combinations used, as well as their instruction orderings. As depicted in Figure 1(b) and Figure 1(d), where the same compiler yet different optimization levels are adopted, the most significant impact on the produced binaries is observed for this short program. The code in Figure 1(d) where no optimization is applied, contains the well-known function prologue of *push rbp* and *mov rsp*, *rbp* that saves the caller's stack frame, and epilogue of *pop rbp* and *retn* that appears at the end of a function for restoring the stack and registers to the state before the function is called, while the code in Figure 1(b) does not use the stack at all but begins with some accesses to the function arguments passed in register *esi* and *edi*. If functions generated by the compiler gcc with O0 setting were known to almost invariably begin with the typical prologue and end with the typical epilogue, while other compiler settings hardly ever did so, then the existence of such typical instruction combinations will serve as significant features to identify compiler optimization settings.

Likewise, as illustrated in Figure 1(b) and Figure 1(c), where two different compilers gcc and clang with the same optimization level are applied, the generated function bodies are of the same length, but different instructions including both operators and operands being adopted to achieve the same functionality. For example, for judging the condition $a > 1$, gcc utilizes *cmp edi*, *1* and *jle*, while clang adopts another instruction combination of *cmp eax*, *2* and *jl*. In addition, differences are shown in the instruction layouts. For example, the starting instruction *cmp esi*, *1* in Figure 1(b) that judges the condition $b == 1$ is arranged as the third instruction in Figure 1(c). The least differences are observed between the codes in Figure 1(b) and Figure 1(e), where two different gcc versions are applied to process the source code. As it shows, their binary codes are almost identical, except that Figure 1(b) adopts the combination of two instructions *mov edx*, *eax* and *sar edx*, *1Fh* to accomplish the same goal (setting all the bits in *edx* to the value of the highest bit in *eax*) as the single instruction cdq adopted in Figure 1(e). Despite not being as remarkably impactful as the compiler

```
0000000000012150 <png_colormap_compose>:
12150    mov    [rsp-28h], rbx
12155    mov    [rsp-20h], rbp
1215A    mov    ebx, r9d
1215D    mov    [rsp-18h], r12
12162    mov    [rsp-10h], r13
12167    mov    rbp, rdi
1216A    mov    [rsp-8], r14
1216F    mov    r13d, r8d
12172    sub    rsp, 28h
12176    mov    r12d, ecx
12179    call   decode_gamma
1217E    mov    esi, r13d
12181    mov    rdi, rbp
12184    mov    edx, ebx
12186    mov    r14d, eax
12189    call   decode_gamma
1218E    mov    edi, 0FFh
12193    sub    edi, r12d
12196    imul   eax, edi
12199    imul   r14d, r12d
1219D    cmp    ebx, 2
121A0    lea    esi, [rax+r14]
121A4    mov    eax, esi
121A6    jz     <png_colormap_compose+A0h>
121A8    mov    rdx, cs:png_sRGB_delta_ptr
121AF    shr    eax, 0Fh
121B2    mov    rcx, cs:png_sRGB_base_ptr
121B9    and    esi, 7FFFh
121BF    movzx  edx, byte ptr [rdx+rax]
121C3    movzx  eax, word ptr [rcx+rax*2]
121C7    imul   edx, esi
121CA    shr    edx, 0Ch
121CD    add    edx, eax
121CF    movzx  eax, dh
121D2    mov    rbx, [rsp+28h-28h]
121D6    mov    rbp, [rsp+28h-20h]
121DB    mov    r12, [rsp+28h-18h]
121E0    mov    r13, [rsp+28h-10h]
121E5    mov    r14, [rsp+28h-8]
121EA    add    rsp, 28h
121EE    retn
121F0    shl    eax, 8
121F3    add    esi, eax
121F5    mov    eax, esi
121F7    shr    eax, 10h
121FA    lea    eax, [rsi+rax+8000h]
12201    shr    eax, 10h
12204    jmp    <png_colormap_compose+82h>
```

(a) Compiled with gcc 4.7 and O2

```
0000000000012000 <png_colormap_compose>
12000    push   r14
12002    push   r13
12004    mov    r14d, r8d
12007    push   r12
12009    push   rbp
1200A    mov    r13, rdi
1200D    push   rbx
1200E    mov    ebp, r9d
12011    mov    r12d, ecx
12014    call   decode_gamma
12019    mov    edx, ebp
1201B    mov    esi, r14d
1201E    mov    rdi, r13
12021    mov    ebx, eax
12023    call   decode_gamma
12028    mov    edx, 0FFh
1202D    sub    edx, r12d
12030    imul   eax, edx
12033    imul   ebx, r12d
12037    add    eax, ebx
12039    cmp    ebp, 2
1203C    mov    edx, eax
1203E    jz     <png_colormap_compose+78h>
12040    mov    rcx, cs:png_sRGB_delta_ptr
12047    shr    edx, 0Fh
1204A    and    eax, 7FFFh
1204F    pop    rbx
12050    pop    encoding
12051    movzx  ecx, byte ptr [rcx+rdx]
12055    pop    alpha
12057    pop    display
12059    pop    background
1205B    imul   eax, ecx
1205E    mov    rcx, cs:png_sRGB_base_ptr
12065    movzx  edx, word ptr [rcx+rdx*2]
12069    shr    eax, 0Ch
1206C    add    eax, edx
1206E    movzx  eax, ah
12071    retn
12078    shl    edx, 8
1207B    add    eax, edx
1207D    mov    edx, eax
1207F    shr    edx, 10h
12082    lea    eax, [rax+rdx+8000h]
12089    pop    rbx
1208A    shr    eax, 10h
1208D    pop    rbp
1208E    pop    r12
12090    pop    r13
12092    pop    r14
12094    retn
```

(b) Compiled with gcc 6.5 and O2

FIGURE 2. The assembly codes for function pngcolormapcompose (from the libpng library) that are produced with two different compiler versions.

family and optimization level act on the produced binaries, compiler version can also enforce significant changes to the produced binary codes, where Figure 2 just gives one such representative example taken from our constructed dataset.

From the discussions above, we can see that clues for inferring the compiler-related settings are well implied in their produced binaries. Yet these clues are scattered all over the binary, and generally subtle and trivial, making them difficult to be systematically abstracted and refined to achieve effective and efficient detection without a deep and laborious large-scale analysis from domain experts. Thus, the manually identified clues or features can either be insufficient or even contrary to what intuition might suggest. For instance, the well-known *push rbp* and *mov rsp*, *rbp* prologue seems to be a good indicator for determining the compiler settings adopted, which, however, turns out not to be in the top ranked idioms [28], [29].

Traditional machine learning based methods generally extract extremely large amount of features with artificially defined templates, and reduce them with certain feature selection strategies to ensure scalability. Without a comprehensive understanding on the characteristics of the binary code, the compiler and the programming language, the manually-crafted feature extraction and selection strategies tend to either capture massive irrelevant and redundant features that make the whole compiler identification approach non-scalable, or fail to extract subtle yet significant ones that lead to unreliable identification results. Inspired by the tremendous successes and superior feature learning power of deep learning in various program analysis tasks [16], [23], [34], [45], [48], [51], in this work, we resort to typical neural network structures to automatically capture and select the scattered, subtle yet significant features that manifest compiler settings, so as to achieve less human intervened yet effective and efficient fine-grained compiler identification.

B. PROBLEM DEFINITION

We formally define the compiler identification task we are going to solve as follows.

Definition 1 (Fine-Grained Compiler Identification): Given an individual function f in its binary form and stripped of debug and symbol information, we infer from it the compiler settings $\mathcal{D}$ adopted in the compilation phase that produces it, with a set of models $\mathcal{M}$ which are sought in a learning way.

The fine-grain in Definition 1 that we want to achieve reflects in two aspects. Firstly, the object to be operated on is an individual function rather than a whole program [6], [12], [39], [45]. We refer to the individual function as an independent function, about which we know nothing else (such as its adjacency functions in the call graph) but just the function itself. It does not distinguish between compiler-related [27] or user-defined functions, which means a compiler-related function (i.e. compiler helper/utility functions) can also be an individual function. Secondly, the compiler settings $\mathcal{D}$ to be revealed contain more elements including the compiler family, optimization level, compiler version or combinations of them (such as both compiler family and optimization level) than simple compiler family [23], [29] or optimization level [14]. That is, when we say a model $m \in \mathcal{M}$ infers compiler optimization level, we know in prior the compiler family of the binary to be detected. For example, given a binary function f compiled with gcc, we would choose the model trained on gcc-compiled (with different optimization levels) samples to identify whether it is compiled with optimization level O0 or O2. When we say a model $m \in \mathcal{M}$ infers both compiler family and optimization level, such a model can determine simultaneously whether f is compiled with gcc and O1 or clang and O3. Apparently, depending on the specific component in $\mathcal{D}$ to be revealed, the difficulty of training an accurate model $m \in \mathcal{M}$ varies, as different amounts

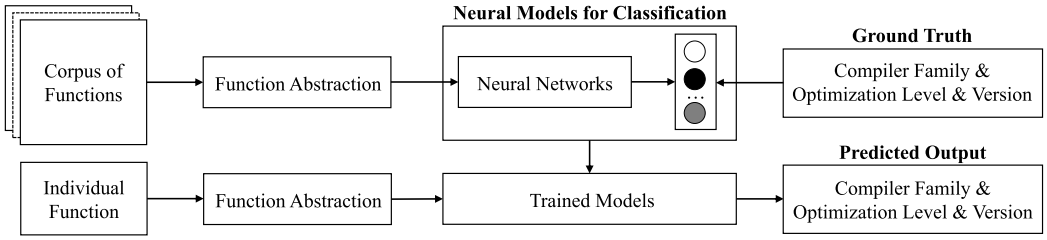


FIGURE 3. The basic framework of NeuralCI.

of clues or footprints are preserved in the finally produced binaries under different compiler settings. Intuitively and as confirmed in our experimental evaluations, the difficulty of learning an model that accurately works for compiler identification increases in the order $\mathcal{M}_\psi < \mathcal{M}_o < \mathcal{M}_\upsilon < \mathcal{M}_{combo}$, where $\mathcal{M}_\psi$, $\mathcal{M}_o$, $\mathcal{M}_\upsilon$ denote the models trained for detecting compiler family, optimization level, and compiler version solely, and $\mathcal{M}_{combo}$ denotes the model for detecting compiler setting combinations.

C. OUR DESIGN OVERVIEW

The overview of our designed NeuralCI to solve the above discussed compiler identification problem is depicted in Figure 3, which consists of two phases: the training phase (upper figure), and the detection phase (lower figure). The training phase includes three steps. As a deep learning-based method, the first step is to construct a high-quality dataset comprised of labeled functions which will be discussed in Section III. The second step takes each raw function as input and outputs a normalized instruction sequence via a lightweight abstraction strategy implemented in the function abstraction module. Then these normalized sequences together with their ground truth labels are fed into the neural network-based classification module to train compiler identification models. The detection phase reads in an individual function, processes it with the function abstraction and utilizes the trained models to produce predictions. In the following sections, we discuss the details of the function abstraction module and the neural network-based classification module, respectively. Table 1 shows some important notations used in this paper.

III. FUNCTION ABSTRACTION

A function must be represented in certain forms such that it can be processed for further analysis [16], [23], [34], [48], [51]. The typical ways include using the raw byte sequence, the assembly instruction sequence, or the control flow graph [23] to depict a function. As discussed in Section II-A and many existing works [12], [37], [38], [51], different compiler settings generally produce distinguishable assembly instructions in their specific instruction contents, orderings, and combinations. Also, as indicated by the outstanding compiler family identification accuracy in [28], short assembly instruction sequences successfully capture compiler-related

TABLE 1. Notations.

Notation	Description
f	A function
ins	An assembly instruction
e	The word embedding vector of an instruction
A	A plain feature matrix corresponding to an instruction sequence
d	Dimension of e
Q	The query matrix as specified in Transformer's attention
K	The key matrix as specified in Transformer's attention
V	The value matrix as specified in Transformer's attention
$\sqrt{d_k}$	A scaling factor as specified in Transformer's attention
g	An attention vector
h	The concatenated hidden state vector of the BiGRU layer
μ_w	An external context vector specified in the query-based attention
σ	Element-wise sigmoid function

features. Thus, in this work we choose to use the assembly instruction sequence within the function body as the representation of each function and use IDA Pro for the parsing.[1] That is, a function f is represented as a sequence $S_f = \{ins_1, ins_2, \cdots, ins_n\}$, where n denotes the number of instructions within the function, and each assembly instruction ins_i consists of an opcode (i.e. mnemonic) and an ordered list of operands. Also, the sequence ensures that $\forall i, j \in [1, n]$, $addr\left(ins_j\right) > addr\left(ins_i\right)$ if $j > i$, where $addr\left(ins_i\right)$ returns the address of ins_i.

However, many existing binary analysis tasks [23], [51] suggested that it is unwise to work directly on raw assembly instructions. For our case, we want to capture features reflecting the compiler details rather than the function functionality. Using all instructions as they appear exactly in the functions may immerse us in too many details of the functionalities, which may accordingly increase the complexity of representation learning and decrease the embedding quality (as too many different instructions are preserved) in the instruction embedding phase, and distract the attention of successive neural network training as well. On the other hand, excessive normalization to the instructions will introduce a mass of unintentional human bias, and lead to the loss of certain subtle yet significant features, such as the instance of the different ways of gcc and clang adopted to handle the $a > 1$ predicate.

[1] We assume a reliable way to identify function and instruction boundaries, as well as 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 beyond the scope of this paper.

To this end, we choose to process the raw instructions with a lightweight abstraction strategy. Specifically, we perform normalization on 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 isolated immediates[2] with absolute values below certain threshold (which is set to 5,000 in current design) remain intact, while in all other cases the immediates in the operands are substituted with the symbol *IMM*.

The first two rules are devised in consideration of the significant differences caused by different compiler settings to the instructions' operators and operands of register type, which apparently form good indicators (thus should not be weakened) for inferring the settings. The third rule is to omit meaningless displacements that appear solely in the operand of the mov instruction for direct addressing. For instance, the instruction "*mov ebx, [0×3435422]*" will be transformed to "*mov ebx, MEM*" according to the first three rules. The fourth rule is to avoid considering those unnecessary details while still retain subtle yet significant ones. For example, as shown in Figure 1(d) where the first several instructions "*mov [rbp-24h], edi*; *mov [rbp-28h], esi*; *mov eax, [rbp-28h]*" perform stack operations, there is no need for us to care about the specific positions of the stack variables to be set or accessed, and the fourth rule enables us to ignore them by simplifying these instructions to "*mov [rbp-IMM], edi*; *mov [rbp-IMM], esi*; *mov eax, [rbp-IMM]*". Meanwhile, the isolated immediate and threshold setting help us capture subtle clues as discussed in the previous case of gcc and clang when handling the predicate. Together with rule 1 and rule 2, rule 4 will transform the gcc instruction combinations "*cmp edi, 1*; *jle 4005C5*" to "*cmp edi, 1*; *jle IMM*", and the clang instruction combinations "*cmp eax, 2*; *jl 40054D*" to "*cmp eax, 2*; *jl IMM*".

IV. NEURAL MODELS FOR COMPILER IDENTIFICATION

A. INSTRUCTION EMBEDDING

As we adopt a learning model for detecting compiler settings, the normalized assembly instruction sequences must be transformed to numerical vectors such that they are able to be fed as inputs to the subsequent classifiers. Since our design chooses to leverage the advanced deep neural networks to grasp significant compiler-setting-related patterns, the straightforward BoW (bag of word) representation fails to work. As such, we use word embedding to distribute a vector for each unique instruction first, based on which the whole instruction sequence can then be modeled and represented.

There are several word embedding choices that can be leveraged for our application, of which the one-hot encoding has been widely deployed. It represents each unique word by a n-dimensional vector, with the i^{th} dimension being set to 1 and all other dimensions being set to 0, where i is the index of the word in the vocabulary of size n. This technique is computationally intractable as the generated vectors are too sparse (the same dimension as the size of the whole vocabulary) and generally needs to do joint-learning with subsequent neural networks, making the learnt word semantics significantly task-specific. In this respect, NeuralCI leverages the popular skip-gram model [24] to learn more compact vector representations that carry instruction co-occurrence relationships and lexical semantics in an independent and unsupervised manner, so as to make the learnt vectors reusable in other binary analysis tasks [16], [19], [48], [51]. Specifically, we treat each basic block as a sentence and each normalized instruction within the basic block as a word, and feed all basic blocks from our binary collection to the skip-gram model to learn for each unique instruction a d-dimensional vector, by minimizing the loss of observing an instruction's neighborhood (within a window w) conditioned on its current embedding. The objective function of skip-gram can be defined as [26]:

$$\arg\min_{\phi} \sum_{-w\le j\le w, i\neq j} -\log p(ins_{i+j}|\phi(ins_i)), \tag{1}$$

where $\phi(ins_i)$ is the current embedding of e_i. We train the embedding model for 100 epochs with the learning rate of 0.001 and context window size w of 5.

B. NEURAL NETWORK MODELS

Based on the instruction embeddings learned by skip-gram, it is promising to explore different schemes such as max-pooling, averaging or concatenation to aggregate the embeddings for each normalized instruction sequence, and then feed it 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 correlations, but also global or long-range instruction dependency; in this respect, it calls for sequence learning models to better capture the expressive compiler-specific patterns and features from instruction sequences for compiler identification. As advanced neural network structures, both CNN and RNN have boosted the state-of-the-arts in sequence learning. As such, in this work, NeuralCI attempts both CNN and RNN models to learn the semantic and structural information of instruction sequences and leverages these advances to identify their compiler settings.

1) CNN-BASED MODELS

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.

[2]When there is nothing else but an immediate in the operand, we refer to this immediate as isolated.

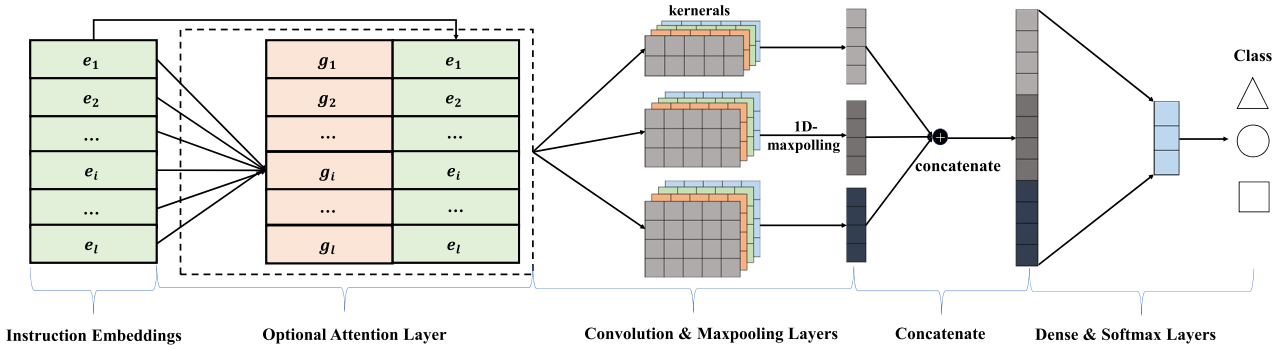


FIGURE 4. The architecture of our CNN-based models in NeuralCI.

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-based models is shown in Figure 4. On the basis of instruction embedding, each instruction sequence is first transformed into a raw feature matrix $A \in \mathbb{R}^{l\times d}$ where l is the sequence length,[3] and d is the instruction embedding dimension.

$$A = [e_1, e_2, \cdots, e_i, \cdots, e_l]^T, \tag{2}$$

where $e_i \in \mathbb{R}^d$ is the corresponding embedding of ins_i in the sequence. Then, in the convolutional layer, k convolutional filters with shape $n \times d$ are adopted for convolution operations on the raw feature matrix A, obtaining a new feature matrix $c \in \mathbb{R}^{(l-n+1)\times k}$, where n denotes the kernel size. To extract different views of feature patterns, we convolute A by different kernels of size 2, 3 and 4 (analogy to 2, 3 and 4 instruction grams respectively), which are then passed through 1D-maxpooling layer for dimensionality reduction. The resulting representations are finally concatenated through a dense layer to be fed to a softmax layer for compiler prediction.

Impressed by the performance gains of integrating attention into neural network structures in many natural language processing (NLP) tasks, we further attempt introducing an attention layer between the input layer and the convolution layer as depicted in Figure 4. In our case, the intuition of introducing such an attention layer is to capture long term contextual information and correlation between non-consecutive instructions. Specifically, NeuralCI implements two kinds of attentions, including the scaled dot product attention and an additive attention, which are originally used to estimate the relevance between hidden vectors.

[3]The actual number of instructions varies across functions, with some of them containing instructions more or less than l. As existing works do, we either zero-pad or truncate that sequence to length l.

The scaled dot product attention is the attention mechanism proposed in the famous Transformer [42], which computes attentions simultaneously with matrix operations:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{3}$$

where $\text{Attention}(Q, K, V) \in \mathbb{R}^{l\times d_v}$ is the attention matrix in which each row g_i is a d_v-dimensional attention vector for the corresponding row in the input matrix, $Q \in \mathbb{R}^{l\times d_k}$, $K \in \mathbb{R}^{m\times d_k}$ and $V \in \mathbb{R}^{m\times d_v}$ are three matrices derived by multiplying the input matrix with weight matrices, and $\sqrt{d_k}$ is a scaling factor to ensure more stable gradients.

The additive attention mechanism we adopt is the one originally used to enhance sequence tagging [50], based on which we calculate a context vector g_i for each (instruction) embedding e_i in matrix A as:

$$g_i = \sum_{j=1}^{l} \alpha_{i,j} \cdot e_j, \tag{4}$$

$$\alpha_{i,j} = \frac{\exp\left(\beta_{i,j}\right)}{\sum_{j'} \exp\left(\beta_{i,j'}\right)}, \tag{5}$$

$$\beta_{i,j} = \sigma\left(W_z h_{i,j} + b_z\right), \tag{6}$$

$$h_{i,j} = \tanh\left(W_x e_i + W_y e_j + b_x\right), \tag{7}$$

where, $\beta_{i,j}$ captures the correlation between the instruction embedding e_i and e_j; $\alpha_{i,j}$ is the attention weight normalized with Softmax function so as to ensure $\alpha_{i,j} \geq 0$ and $\sum_j \alpha_{i,j} = 1$; σ is the element-wise sigmoid function; W_x,W_y and W_z are respectively the weight matrices corresponding to e_i, e_j and their non-linear combinations; b_x and b_y are the bias vectors.

For both attention mechanisms, NeuralCI concatenates the vectors output from the attention layer with the initial embedding vectors as a new representation to be further processed with the convolution and subsequent layers. For simplicity, we use $Neural^{BS}_{CNN}$, $Neural^{SD}_{CNN}$, and $Neural^{AD}_{CNN}$ to specify the base CNN, the enhanced CNN with scaled dot product attention and the enhanced CNN with additive attention throughout the experimental evaluations.

2) RNN-BASED MODELS

RNN is known to learn the sequential dependency, and strict to align the positions and contexts for the instances in the input sequences. By capturing long range dependencies between the instructions in a sequence, RNN can produce a comprehensive and contextualized latent vector representation for the entire instruction sequence. In consideration of the vanishing/exploding gradient issue faced by naïve RNN, NeuralCI employs the Gated Recurrent Unit (GRU) [15], [17] in the current design to alleviate the problem. Figure 5 depicts the basic structure of our RNN-based models. Similar to the CNN-based models, the matrix A composed of instruction embeddings forms the input, which is read through a GRU cell such that various non-linear transformations are performed to generate a hidden vector state h_i at each timestep i. Improved over GRU, bidirectional GRU (BiGRU), which leverages two GRU cells reverse to each other to process the sequence, captures both the previous and the future timestep features via forward and backward states. Therefore, NeuralCI adopts the BiGRU structure to jointly capture both the forward and backward sequential dependency and global contextual information in the instruction sequence, so that the hidden state vector h_i at timestep i can be concatenated as:

$$h_i = \left[\overrightarrow{h_i}\,;\,\overleftarrow{h_i}\right]. \tag{8}$$

After forward and backward reading the entire input sequence, in Figure 5(a) the hidden states h_l corresponding to the last timestep will act as the latent vector representation in our naïve RNN model, which is then fed to subsequent dense and Softmax layers for predication. During the training process, the training loss is adopted to measure the correctness of sequence learning and compiler prediction. Also, dropout is applied to prevent the neural network from overfitting.

Alternatively, as illustrated in Figure 5(b) and 5(c), NeuralCI also attempts introducing an attention layer between the GRU layer and the dense layer on the basis of our naïve RNN model, in the sense of attending to partial informative instructions (that are potentially more important for compiler identification) rather than focusing on all instructions equally. In Figure 5(b), NeuralCI aggregates the attention vectors produced by either of the two attention mechanisms discussed in CNN-based models) with an average pooling[4] operation, generating a single informative vector representation for a sequence. Besides these two attention mechanisms, as shown in Figure 5(c), NeuralCI also incorporates another kind of attention mechanism [46] which introduces an external context/query vector μ_w to compute for each word (embedding) in a sentence an importance weight and then compute a weighted sum of the word embeddings based on the importance weights as the sentence's vector representation. For our application, we compute a vector for the instruction sequence with this attention mechanism as follows:

$$g = \sum_{i=1}^{l} \alpha_i \cdot h_i, \tag{9}$$

$$\alpha_i = \frac{\exp\left(\beta_i^T \mu_w\right)}{\sum_i \exp\left(\beta_i^T \mu_w\right)}, \tag{10}$$

$$\beta_i = \tanh\left(W_x h_i + b_x\right) \tag{11}$$

where, α_i is the normalized attention weights through Softmax function; W_x is the weight matrix corresponding to h_i, b_x is the bias vector, both of which are to be jointly learned with the context vector μ_w during the training process. The same as the naïve RNN, the latent vector produced with either of the three attention mechanisms is further fed to subsequent dense and Softmax layers for compiler prediction. Also, to get these RNN models distinguishable, we use $Neural_{GRU}^{BS}$, $Neural_{GRU}^{SD}$, $Neural_{GRU}^{AD}$ and $Neural_{GRU}^{QU}$ (the most impressive characteristic of the third attention mechanism is the introduction of an external query vector, thus we call it QUery-based-attention for short) to depict them for simplicity in the following experimental evaluations.

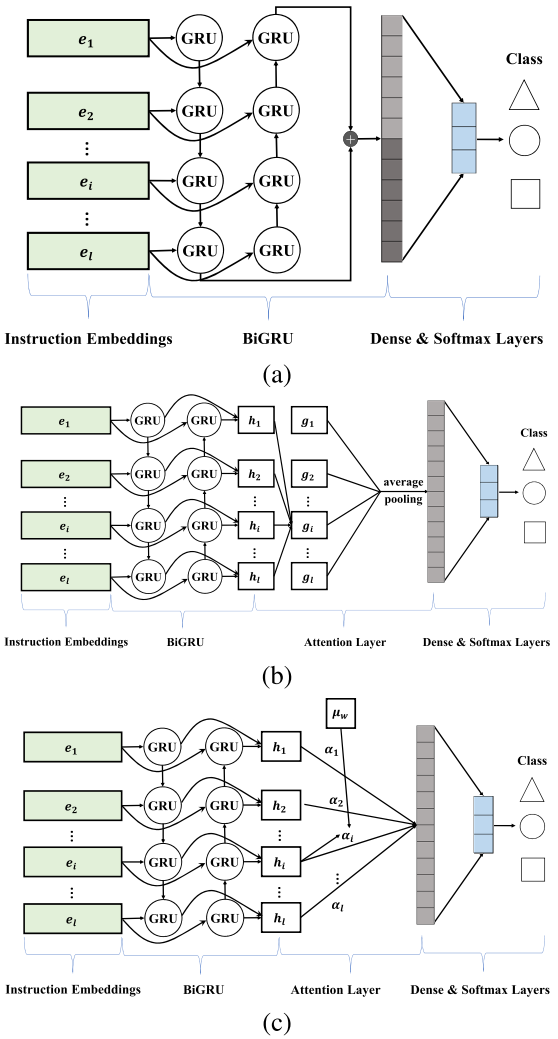


FIGURE 5. The architecture of our RNN-based models in NeuralCI.

[4]Max pooling also works without obvious difference in performance compared with average pooling.

V. EXPERIMENTS AND EVALUATION

A. DATASET CONSTRUCTION

To evaluate the performance of NeuralCI, we collected 19 widely used C/C++ open source projects (including binutils 2.32, busybox 1.31.0, ccv 0.7, coreutils 8.31, curl 7.65.3, ffmpeg 4.2.1, FreeImage 3.17.0, gdb 8.2, git 2.22.0, gsl 2.6, zlib 1.2.11, libhttpd 2.0, libpng 1.6.34, openmpi 3.1.1, openssl 1.1.1c, postgresql 10.4, sqlite 3.22, valgrind 3.15, and vim 8.1) as the basics to construct the dataset. To be specific, we process these projects with the following steps:

- Three different compilers involving multiple versions including GCC (4.7, 4.8, 4.9, 5.5, 6.5, 7.4), Clang (3.8, 5.0) and ICC 19.0, 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.
- During the training phase, to avoid neural models seeing functions that are really similar to the ones in the testing phase, which, if not properly coped with, can inflate the performance metrics, we only keep unique functions. Specifically, a function is considered redundant if it has the same normalized instructions as others. 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 854,858 unique functions from 4,810 binaries[5] with an average of about 260 instructions within a function. The distribution of the function sizes is depicted in Figure 6, where 50% of functions contain less than 100 instructions, and nearly 90% of functions contain less than 500 instructions.

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 and their raw assembly instructions. The function abstraction module is implemented in Java, and the neural modeling module is implemented using Python and Tensorflow framework. The skip-gram implementation provided in gensim [4] is used to generate instruction embedding vectors, with the embedding size d setting to 100. The maximum length l for an instruction sequence is set to 500, which ensures to cover the complete semantics of nearly 90% functions in our dataset. The number of convolution filters in the CNN-based models is set to 128, and so is the dimension of the GRU hidden state vectors.

[5]It should be noted that not every project can be successfully compiled under every compiler setting. Also, the successful compilation of different projects produce varying number of binaries.

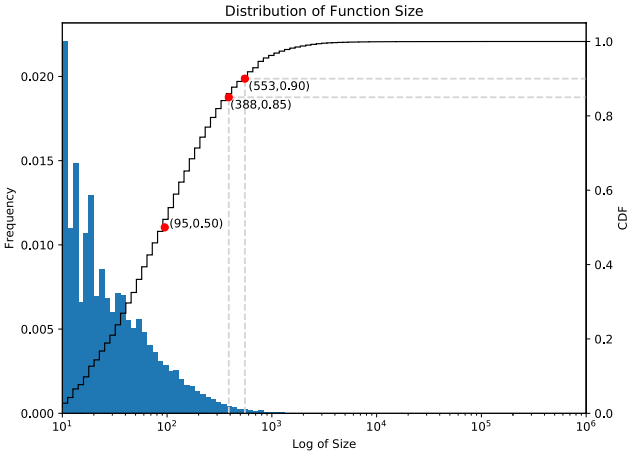


FIGURE 6. Distribution of the function sizes.

For experimental settings, we randomly split the dataset into training, validation and testing sets according to a percentage of 80%, 10% and 10% respectively. The neural network models are trained with a Tesla V100 GPU card using a batch size of 128, initial learning rate 0.001 (we divide the learning rate by 10 when the validation loss has stopped improving for at least 5 epochs), and the Adam optimizer. For each epoch, the training samples are shuffled and accuracy on the validation set is calculated. Also, the EarlyStopping mechanism is leveraged to stop the training after the epoch when the validation accuracy rate no longer rises, so as to avoid over-fitting, non-convergence and other problems. Finally, we take the model with the best 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, we evaluate the performance of NeuralCI on identifying the compiler family, optimization level, compiler version and compiler setting combination respectively, and report the comparative results across the neural network models as well as against existing function level methods that support the detection of corresponding compiler settings. Note that the accuracy values in Table 2, 3, 5, 6, 7,

TABLE 2. Compiler Family Identification Results.

Model	Accuracy	Precision	Recall	F1
$Neural^{BS}_{CNN}$	98.5%	0.985	0.985	0.985
$Neural^{SD}_{CNN}$	98.5%	0.985	0.985	0.985
$Neural^{AD}_{CNN}$	98.5%	0.985	0.985	0.985
$Neural^{BS}_{GRU}$	98.6%	0.987	0.986	0.986
$Neural^{SD}_{GRU}$	**98.7%**	**0.987**	**0.987**	**0.987**
$Neural^{AD}_{GRU}$	98.6%	0.986	0.986	0.986
$Neural^{QU}_{GRU}$	98.6%	0.986	0.986	0.986

TABLE 3. Optimization Level Identification Results.

Condensed levels	Model	GCC				Clang				ICC			
		Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1
2-Levels: O_L, O_H	$Neural^{BS}_{CNN}$	98.6%	0.987	0.986	0.986	91.5%	0.914	0.915	0.914	95.9%	0.960	0.959	0.959
	$Neural^{SD}_{CNN}$	98.7%	0.987	0.987	0.987	91.8%	0.918	0.918	0.918	95.7%	0.957	0.957	0.957
	$Neural^{AD}_{CNN}$	98.7%	0.987	0.987	0.987	91.8%	0.918	0.918	0.918	95.7%	0.958	0.957	0.958
	$Neural^{BS}_{GRU}$	98.8%	0.988	0.988	0.988	90.8%	0.909	0.908	0.908	95.1%	0.953	0.951	0.952
	$Neural^{AD}_{GRU}$	98.8%	0.988	0.988	0.988	91.2%	0.913	0.912	0.913	95.3%	0.954	0.953	0.953
	$Neural^{QU}_{GRU}$	98.2%	0.982	0.982	0.982	90.4%	0.902	0.904	0.903	95.0%	0.950	0.950	0.950
3-Levels: O0, O1, O_H	$Neural^{BS}_{CNN}$	98.8%	0.988	0.988	0.988	91.4%	0.916	0.914	0.915	95.9%	0.960	0.960	0.960
	$Neural^{SD}_{CNN}$	98.9%	0.989	0.989	0.989	91.0%	0.911	0.910	0.911	96.0%	0.960	0.960	0.960
	$Neural^{AD}_{CNN}$	98.9%	0.989	0.989	0.989	91.7%	0.919	0.917	0.918	95.5%	0.955	0.955	0.955
	$Neural^{BS}_{GRU}$	98.7%	0.987	0.988	0.987	92.5%	0.927	0.925	0.926	96.3%	0.965	0.963	0.963
	$Neural^{SD}_{GRU}$	98.8%	0.989	0.988	0.988	91.3%	0.916	0.913	0.914	94.9%	0.949	0.949	0.949
	$Neural^{SD}_{GRU}$	98.8%	0.988	0.988	0.988	91.6%	0.919	0.916	0.917	95.7%	0.958	0.957	0.957
	$Neural^{AD}_{GRU}$	98.8%	0.989	0.988	0.988	92.1%	0.925	0.921	0.922	95.6%	0.957	0.956	0.956
	$Neural^{QU}_{GRU}$	98.8%	0.988	0.988	0.988	92.0%	0.923	0.920	0.921	95.5%	0.955	0.955	0.955

and 8 all refer to the total accuracy, and the precision, recall, and f1-score values in these tables all refer to the weighted-average precision, recall and f1-score respectively.

To be specific, let k be the class label, $\{c_1, c_2, \cdots, c_k\}$ be the numbers of samples with respect to each class, and $\{c'_1, c'_2, \cdots, c'_k\}$ be their numbers of correctly classified samples by a classifier. The total accuracy can be defined as:

$$Accuracy = \frac{\sum_{i=1}^{k} c'_i}{\sum_{i=1}^{k} c_i} \tag{12}$$

Let $\{p_1, p_2, \cdots, p_k\}$, $\{r_1, r_2, \cdots, r_k\}$ and $\{f_1, f_2, \cdots, f_k\}$ be the precision, recall and f1-score values computed with respect to the k classes respectively. The weighted-average precision, recall and f1-score can be defined as:

$$Precision = \sum_{i=1}^{k} \frac{c_i}{\sum_{j=1}^{k} c_j} p_i \tag{13}$$

$$Recall = \sum_{i=1}^{k} \frac{c_i}{\sum_{j=1}^{k} c_j} r_i \tag{14}$$

$$F1 = \sum_{i=1}^{k} \frac{c_i}{\sum_{j=1}^{k} c_j} f_i \tag{15}$$

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 V-B. As it shows in Table 2, the NeuralCI models achieve rather high accuracy and f1-score in identifying the compiler family. Also, there exist no obvious performance differences between the NeuralCI models, among which $Neural^{SD}_{GRU}$ exhibits the best performance with an accuracy of 98.7% and f1-score of 0.987 for the compiler family identification task.

TABLE 4. Precision and Recall Values Before and After Separating Out O1 From O_L.

Metrics	3-Level Case			2-Level Case	
	O0	O1	O_H	O_L	O_H
Precision	0.999	0.938	0.816	0.923	0.845
Recall	0.968	0.890	0.910	0.948	0.781

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 provided in works [27], [28] that condenses the 4 optimization levels to 2 classes 'low' and 'high', considering the findings presented in existing studies [27], [28] that it is difficult to distinguish between O2 and O3 compiled binaries. That is, O0 and O1 will be considered as the low optimization class as denoted by O_L, while O2 and O3 will be considered as the high optimization class as denoted by O_H. In addition to this problem simplification strategy, we also test the performance of the models with a 3-level splitting way that condenses the optimization levels to O0, O1 and O_H.

The data summarized in Table 3 shows: 1) The NeuralCI models exhibit relatively good detection results in the either 2-level or 3-level optimization condensation case, with little performance differences observed across the models still. 2) Surprisingly, NeuralCI performs equally or even better on the 3-level optimization option identification task than

TABLE 5. Compiler Version Identification Results.

Model	GCC (Major)				Clang				ICC (Minor)			
	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1
$Neural^{BS}_{CNN}$	94.4%	0.944	0.944	0.944	84.2%	0.844	0.842	0.839	77.8%	0.782	0.778	0.779
$Neural^{SD}_{CNN}$	94.4%	0.943	0.944	0.943	84.1%	0.841	0.841	0.839	77.0%	0.776	0.770	0.771
$Neural^{AD}_{CNN}$	94.4%	0.944	0.944	0.944	84.1%	0.841	0.841	0.839	76.9%	0.774	0.769	0.770
$Neural^{BS}_{GRU}$	93.7%	0.937	0.937	0.936	82.3%	0.821	0.823	0.821	74.7%	0.748	0.747	0.747
$Neural^{SD}_{GRU}$	94.0%	0.940	0.940	0.940	82.3%	0.821	0.823	0.821	75.4%	0.754	0.754	0.754
$Neural^{AD}_{GRU}$	93.7%	0.937	0.937	0.937	82.4%	0.823	0.824	0.822	74.7%	0.747	0.747	0.747
$Neural^{QU}_{GRU}$	93.8%	0.938	0.938	0.938	83.3%	0.834	0.833	0.834	74.7%	0.747	0.747	0.747

TABLE 6. Results for Identifying Compiler Family and Optimization Level Simultaneously.

Case	Model	Accuracy	Precision	Recall	F1
2-Levels: O_L, O_H	$Neural^{BS}_{CNN}$	94.5%	0.944	0.945	0.944
	$Neural^{SD}_{CNN}$	94.8%	0.948	0.948	0.948
	$Neural^{AD}_{CNN}$	94.8%	0.948	0.948	0.948
	$Neural^{BS}_{GRU}$	94.3%	0.943	0.943	0.943
	$Neural^{SD}_{GRU}$	94.6%	0.945	0.946	0.945
	$Neural^{AD}_{GRU}$	95.0%	0.950	0.950	0.950
	$Neural^{QU}_{GRU}$	95.0%	0.951	0.950	0.950
3-Levels: O0, O1, O_H	$Neural^{BS}_{CNN}$	94.8%	0.948	0.948	0.948
	$Neural^{SD}_{CNN}$	95.0%	0.950	0.950	0.949
	$Neural^{AD}_{CNN}$	94.6%	0.946	0.946	0.946
	$Neural^{BS}_{GRU}$	94.9%	0.950	0.949	0.949
	$Neural^{SD}_{GRU}$	95.1%	0.952	0.951	0.951
	$Neural^{AD}_{GRU}$	94.9%	0.950	0.949	0.950
	$Neural^{QU}_{GRU}$	95.1%	0.952	0.951	0.951

TABLE 7. Results For Identifying Compiler Family, (Major) Version and Optimization Level Simultaneously.

Case	Model	Accuracy	Precision	Recall	F1
2-Levels: O_L, O_H	$Neural^{BS}_{CNN}$	83.1%	0.834	0.831	0.829
	$Neural^{SD}_{CNN}$	83.0%	0.830	0.830	0.828
	$Neural^{AD}_{CNN}$	83.1%	0.832	0.831	0.829
	$Neural^{BS}_{GRU}$	81.5%	0.815	0.815	0.815
	$Neural^{SD}_{GRU}$	83.2%	0.833	0.832	0.832
	$Neural^{AD}_{GRU}$	82.9%	0.829	0.829	0.829
	$Neural^{QU}_{GRU}$	82.6%	0.826	0.826	0.824
3-Levels: O0, O1, O_H	$Neural^{BS}_{CNN}$	83.6%	0.838	0.836	0.833
	$Neural^{SD}_{CNN}$	83.7%	0.840	0.837	0.833
	$Neural^{AD}_{CNN}$	83.8%	0.842	0.838	0.835
	$Neural^{BS}_{GRU}$	83.6%	0.838	0.836	0.833
	$Neural^{SD}_{GRU}$	82.7%	0.827	0.827	0.826
	$Neural^{AD}_{GRU}$	82.4%	0.829	0.824	0.820
	$Neural^{QU}_{GRU}$	83.2%	0.831	0.832	0.831

2-level task for some models. As the most obvious example, the largest difference is observed when applying $Neural^{AD}_{GRU}$ to identify the optimization option for Clang, where accuracies of 92.1% and 90.4% are achieved in the 2-level and 3-level case respectively. Table 4 gives their detailed precision and recall values with respect to each optimization level correspondingly. As it shows, by splitting O1 separately out of O_L, the recall of identifying O_H drastically increases from 0.781 to 0.910, indicating that many O_H samples which are falsely classified as belonging to O_L in the 2-level case are now correctly classified again. Meantime, some samples actually belonging to O1 are now falsely classified as O_H, causing the precision of identifying O_H to decrease from 0.845 to 0.816, and the relatively low recall value of 0.89 for the newly separated class O1. This matches the common sense that binaries compiled with O1, O2 or O3 are more similar to each other than to binaries compiled with O0 where no optimization is applied. 3) The performance of NeuralCI in identifying optimization level varies across compilers, with the best and worst performance (about a margin of 7% in accuracy) observed on GCC and Clang respectively, indicating that the differences introduced by different optimization levels of GCC to the produced binaries seem more obvious. Also, the varying detection difficulty of compiler optimization levels implies that different compilers adopt different ways of defining their optimization levels.

3) PERFORMANCE OF IDENTIFYING COMPILER VERSION

Identifying the specific compiler version used to compile a function is intuitively more challenging. Table 5 summarizes the detection results of NeuralCI applied on GCC and Clang that both contain multiple versions in our dataset. As it shows, varying performance of NeuralCI is observed on different compiler family, where better accuracies ranging from 93.7% to 94.5% are achieved on the major versions of GCC compared with the accuracies observed on Clang that range from 82.3% to 84.3%. It suggests that varying rates of "churn" across versions are exhibited in different compiler families,

with GCC producing significantly more varied code between major versions than Clang. Moreover, we evaluate whether NeuralCI is capable of identifying minor compiler versions (GCC 4.7, 4.8 and 4.9). As it shows in the right columns of Table 5, the accuracies are much lower compared with the accuracies achieved in the major version identification case, which conforms to the common sense that more changes are generally introduced across the major version releases than across the minor version releases of a program. The fairish detection accuracies indicate that NeuralCI can capture very subtle yet significant features which may otherwise be missed by artificially crafted feature extraction and selection strategies. Again, no apparent gaps are observed across the neural network models in NeuralCI.

4) PERFORMANCE OF IDENTIFYING COMPILER SETTING COMBINATION

Detecting compiler setting combinations from a binary can be generally achieved by jointly applying multiple models, each of which targets a different part of the settings. For example, to determine both the compiler family and the optimization level for a binary function, we can firstly use the trained model for compiler family identification to detect the family that compiles the function, and then use the optimization level identification model corresponding to the identified family to further detect the optimization level. An alternative way is to train a single model that can detect these settings simultaneously. To check if NeuralCI is competent with this challenging task, we evaluate its performance of identifying compiler family and optimization level altogether first. As the results depicted in Table 6 show, all the models in NeuralCI exhibit relatively good and similar detection performance. Further, Table 7 reports the results of NeuralCI in identifying all compiler setting combination[6] simultaneously, where accuracies ranging between 81.5% and 83.8% are observed across different models.

5) COMPARISONS WITH EXISTING METHODS

In this section, we compare NeuralCI with existing methods including Idioms [29] and Graphlets [28] that support compiler identification on individual functions. Specifically, we implement three methods based on the ORIGIN [10] code shared on github that extract idiom or graphlet features from binary functions, select significant ones via mutual information computation, and train classifiers with the support vector machine (SVM)[7] on the selected idiom features, the graphlet features, and both of them, producing three models for each specific compiler identification task, which we denote as IDM, GRA, and IDM-GRA respectively. Similarly, we choose the top 20,000 features with the highest mutual information scores as significant features and use linear SVMs considering their good performance on high-dimension data sets. Despite that, due to the natural properties of SVM, these methods still fail to scale to our whole dataset by either consuming the whole memory or the training never ends. To deal with that, we randomly select 100,000 samples from the whole dataset, taking 80% of them as the training data and the rest as the test data. We repeat the above process using 10-fold cross validation[8] to obtain averaged evaluation results.

Table 8 summarizes the experimental results, where the values in the column NeuralCI report the best performance achieved by our models. As it shows, NeuralCI outperforms the other methods comprehensively in terms of detection accuracy, precision, recall, f1 and f0.5 values in almost every compiler identification task, except the task of identifying compiler ICC's optimization level where IDM-GRA outperforms NeuralCI by a small margin.

VI. DISCUSSION

Similar to existing binary analysis methods that work on the function level, NeuralCI also assumes the reliable acquisition of functions from the binaries to be analyzed. In this paper, we select the state-of-the-art commercial reverse engineering tool IDA Pro for parsing, in consideration of its performance and wide application in the binary analysis domain. Alternatives, such as Binary Ninjia [2], Radare2 [9], Dyninst [3], Angr [1], etc., are also compatible with NeuralCI. But in general, correct parsing of binaries is still a complex and open issue. To ensure a more reliable way of obtaining functions, NeuralCI might benefit from recent advances that adopt deep learning to identify functions from binaries [34].

In addition to the impacts from the limitations of binary parsing tools, NeuralCI may also suffer from various kinds of code obfuscations [18], [31] enforced on the code to be analyzed. On one hand, code obfuscation techniques such as compression and encryption [41] may disable the analysis at the first step, as they interrupt the correct parsing and extraction of functions from binaries, which is the fundamental basis for NeuralCI and other function-level compiler identification methods. On the other hand, code obfuscation techniques, such as instruction replacement, dead code insertion, function inlining, etc., that commit changes to the code [30], [32] are very likely to destroy or submerge features important to compiler identification, hence decreasing the detection accuracy. One possible solution to this problem is to deobfuscate the binaries first with deobfuscation techniques [22], [44], [49] before enforcing compiler identification. Another possible way is to adopt the idea of adversarial training [40], which trains the compiler identification model with adversarial examples (i.e. obfuscated samples), to improve the resistance against code obfuscation

[6]We report the results on the major compiler versions, considering that NeuralCI does not perform so good on identifying compiler minor versions (as shown in Table 5).

[7]Besides the SVM model, CRF model which incorporates the adjacency between functions is also adopted in the compared works [29] and [28]. Yet it assumes a sequence of functions rather than treating each function independently, we only compare with their SVM-based model that operates on each independent function.

[8]It differs slightly from the standard 10-fold cross validation in that we construct 10 random subsets, each of which contains 100,000 samples randomly selected from the whole dataset.

TABLE 8. Performance Comparison with Existing Compiler Identification Methods.

Task	Metric	NeuralCI	IDM	GRA	IDM-GRA
Compiler	Accuracy	**98.7%**	93.9%	84.2%	94.0%
	Precision	**0.987**	0.943	0.845	0.944
	Recall	**0.987**	0.939	0.842	0.940
	F1	**0.987**	0.938	0.838	0.939
2-Level Optimization Level (GCC)	Accuracy	**98.8%**	97.0%	94.0%	97.2%
	Precision	**0.988**	0.971	0.941	0.973
	Recall	**0.988**	0.970	0.940	0.972
	F1	**0.988**	0.970	0.941	0.972
2-Level Optimization Level (Clang)	Accuracy	**91.8%**	86.5%	85.8%	87.1%
	Precision	**0.918**	0.864	0.854	0.870
	Recall	**0.918**	0.865	0.858	0.871
	F1	**0.918**	0.855	0.849	0.862
2-Level Optimization Level (ICC)	Accuracy	95.9%	97.2%	97.0%	**97.6%**
	Precision	0.960	0.972	0.970	**0.976**
	Recall	0.959	0.972	0.970	**0.976**
	F1	0.959	0.972	0.970	**0.976**
3-Level Optimization Level (GCC)	Accuracy	**98.9%**	96.6%	92.0%	96.8%
	Precision	**0.989**	0.967	0.922	0.969
	Recall	**0.989**	0.966	0.920	0.968
	F1	**0.989**	0.966	0.920	0.968
3-Level Optimization Level (Clang)	Accuracy	**92.5%**	85.7%	82.5%	86.4%
	Precision	**0.927**	0.862	0.829	0.868
	Recall	**0.925**	0.857	0.825	0.864
	F1	**0.926**	0.849	0.821	0.858
3-Level Optimization Level (ICC)	Accuracy	96.3%	97.0%	95.2%	**97.4%**
	Precision	0.965	0.970	0.953	**0.974**
	Recall	0.963	0.970	0.952	**0.974**
	F1	0.963	0.969	0.952	**0.974**
3-Level Compiler Version (Clang)	Accuracy	**84.2%**	81.9%	78.0%	83.8%
	Precision	**0.844**	0.827	0.775	0.845
	Recall	**0.842**	0.819	0.780	0.838
	F1	**0.839**	0.806	0.762	0.828
3-Level Compiler Version (GCC Major)	Accuracy	**94.4%**	89.2%	85.5%	90.5%
	Precision	**0.944**	0.895	0.858	0.908
	Recall	**0.944**	0.892	0.855	0.905
	F1	**0.944**	0.890	0.852	0.903
3-Level Compiler Version (GCC Minor)	Accuracy	**77.8%**	74.0%	62.9%	75.9%
	Precision	**0.782**	0.770	0.635	0.778
	Recall	**0.778**	0.740	0.629	0.759
	F1	**0.779**	0.741	0.630	0.760
Family and Optimization Level (2-Level)	Accuracy	**95.0%**	87.8%	77.3%	88.3%
	Precision	**0.950**	0.879	0.787	0.884
	Recall	**0.950**	0.878	0.773	0.883
	F1	**0.950**	0.872	0.772	0.878
Family and Optimization Level (3-Level)	Accuracy	**95.1%**	87.2%	75.8%	87.7%
	Precision	**0.952**	0.875	0.771	0.880
	Recall	**0.951**	0.872	0.758	0.877
	F1	**0.951**	0.867	0.753	0.873
All (2-Level)	Accuracy	**83.2%**	73.1%	64.4%	74.2%
	Precision	**0.833**	0.750	0.661	0.761
	Recall	**0.832**	0.731	0.644	0.742
	F1	**0.832**	0.724	0.634	0.736
All (3-Level)	Accuracy	83.8%	72.9%	63.2%	74.4%
	Precision	**0.842**	0.757	0.654	0.771
	Recall	**0.838**	0.729	0.632	0.744
	F1	**0.835**	0.726	0.629	0.744

attacks. Overall, how to achieve robust compiler identification in the face of code obfuscation is a challenging yet important research topic, and we leave it as our future work.

To facilitate other researchers conducting experiments and presenting their findings, we have made public the dataset and NeuralCI's source code on github. To construct a relatively reliable dataset, the compilation itself is a challenging task, as the source files usually have numerous dependencies that complicates the compilation process. Worse still, we must ensure the projects are correctly compiled with the specified compiler settings. Finally, 19 projects with relatively complete compilation environments are processed, which costs us a massive of time reading the guidelines accompanied by each project for configuring and compiling them with the specified compiler settings, so as to produce correctly labeled samples. One concern as to the dataset is its diversity that correlates to the generalization ability of the trained models. For this issue, we try to select projects that involve different program types and application domains. Another concern is the effectiveness of NeuralCI on other compilers

or compiler settings. For example, as the projects used to construct our dataset are widely used open source projects that work under the linux system, it's difficult to successfully compile them with the msvc compiler that generally works under the windows system. Thus we did not test NeuralCI on identifying msvc compiled binaries. Besides, there are other compilers that can produce binaries, such as the IBM XL compiler [5]. Covering all of them apparently will promote the usability of the compiler identification researches, but it is a non-trivial task that requires lots of human efforts. We made public our constructed dataset as the first step, with the expectation for more researchers of participating in the compiler identification and related investigations, and together with other researchers we can gradually refine and enrich the dataset with more samples involving more kinds of compilers and programs.

We did not perform a systematic hyper-parameter tuning, but rather adopt either the default or the commonly used empirical values for the hyper-parameters. As illustrated by the evaluation results, NeuralCI trained with current parameter setting achieves very promising detection accuracy. A systematic or exhaustive grid search based hyper-parameter tuning may further improve the NeuralCI's performance, meanwhile consuming a lot more computing resources and time. We leave it as our future work as well.

VII. RELATED WORK

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

A. SIGNATURE MATCHING BASED METHODS

The signature matching based methods [6]–[8] search the binary program against a corpus of generic and rigid signatures, 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 [6], PEiD [8] and LANGUAGE 2000 [7], with its high detection efficiency and low cost. Their drawbacks lie in the stringent expertise 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. MACHINE 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 implicit features of the resulting binaries reflect design and implementation decisions of a certain compiler that is 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 [29] adopting this type of approach was conducted by Rosenblum *et al.* that 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 for the program binaries. High accuracy was observed for inferring the compiler families, but we have no idea of its performance on optimization levels identification as no evaluation was conducted. ORIGIN [28] 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 learned 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 [12], [39]. However, for each individual compiler family, a corresponding separate model needs to be learned. Also, these models do not extract information regarding the optimization levels. To improve efficiency in terms of computational resources and detection time, BinComp [11], [27] adopted a stratified approach to infer different compiler details on different granularity. It identifies compiler family for the whole program via matching of signatures, and conducts compiler version and optimization level detection for compiler-related functions. However, as the compiler-related functions usually constitute a small portion of all functions in real-world programs where user-defined functions hold the principal status, the method does not solve the function-level compiler identification problem which we are targeting in this paper. Basically, accuracy of these machine learning based methods greatly depends on the quality of expert-defined feature extraction templates and feature selection strategies, where more 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], [23], [34], [45], [48], [51]. BinEye [45] is one of the few works that utilize neural network models to implement 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 compiler family and optimization level for each individual function by adopting an abstraction strategy that operates on assembly instructions rather than the raw bytes. o-glassesX [25] designs an attention augmented CNN-based model for compiler identification, where positional encoding is also applied to capture features of instruction orders. Structure2Vec [23] 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 comprehensive detection of the optimization level, the compiler version, and the compiler setting combination in addition to the compiler family as done by Structure2Vec.

VIII. CONCLUSION

In this work, we model functions as normalized instruction sequences using a lightweight abstraction strategy, which are then fed to well-designed neural networks to solve the problem of fine-grained acompiler identification. We implement the methods in a prototype tool NeuralCI, and get its performance evaluated on a large dataset consisting of totally 854,858 unique functions. As the experimental evaluation shows, NeuralCI outperforms existing function level compiler identification methods. The outstanding accuracies reported on detecting the compiler family, optimization level, compiler version and the compiler setting combination strongly suggest that deep neural networks are capable of capturing subtle yet significant features indicative of compiler settings, and sever as a promising and reliable way to reveal these compiler settings in a less-human-and-domain-knowledge-involving manner.

Despite several popular attention mechanisms that work well on NLP tasks are adopted to enhance our models, limited advances are observed. This indicates that the subtle clues implied in a sequence has already been fully tapped by our models. Besides the instructions and their orders, different compiler settings can also cause significant changes to the binary functions' structures. In consideration of that, we plan to model a function with more comprehensive representation forms such as CFG that preserves both syntactic and structural information, and resort to graph neural networks to achieve the performance enhancement for compiler identification task. The explicability of the models is another work we are planning to focus on, with which we may find some interesting patterns to provide heuristics for researchers in the domain. There exist some candidate techniques [21], [35], [36] such as the gradient based saliency map, the regression model based LEMNA etc. that can be referred to. Also, the well designed neural networks that show advanced performance on tasks such as video deblurring [47] are promising to be applied on compiler provenance task after proper adjustment.

ACKNOWLEDGMENT

This work extends their previous conference paper published at SEKE.

REFERENCES

[1] *ANGR*. Accessed: Mar. 27, 2021. [Online]. Available: https://github.com/angr
[2] *Binaryninja*. Accessed: Mar. 27, 2021. [Online]. Available: https://binary.ninja/
[3] *Dyninst*. Accessed: Mar. 27, 2021. [Online]. Available: https://www.dyninst.org/
[4] *GenSim*. Accessed: Mar. 27, 2021. [Online]. Available: https://radimrehurek.com/gensim/models/word2vec.html
[5] *IBM C and C++ Compiler Family*. Accessed: Mar. 27, 2021. [Online]. Available: https://www.ibm.com/products/c-and-c-plus-plus-compiler-family
[6] *IDA*. Accessed: Mar. 27, 2021. [Online]. Available: https://www.hexrays.com/products/ida/index.shtml
[7] (2000). *Language*. [Online]. Available: https://farrokhi.net/language/
[8] *Peid*. Accessed: Mar. 27, 2021. [Online]. Available: https://www.aldeid.com/wiki/PEiD
[9] *Radare2*. Accessed: Mar. 27, 2021. [Online]. Available: https://github.com/radareorg/radare2
[10] *ToolChain-Origin*. Accessed: Mar. 27, 2021. [Online]. Available: https://github.com/dyninst/toolchain-origin
[11] S. Alrabaee, M. Debbabi, P. Shirani, L. Wang, A. Youssef, A. Rahimian, L. Nouh, D. Mouheb, H. Huang, and A. Hanna, *Compiler Provenance Attribution*. Cham, Switzerland: Springer, 2020, pp. 45–78.
[12] T. H. Austin, E. Filiol, S. Josse, and M. Stamp, "Exploring hidden Markov models for virus analysis: A semantic approach," in *Proc. 46th Hawaii Int. Conf. Syst. Sci.*, Jan. 2013, pp. 5039–5048.
[13] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, Dec. 2016, pp. 189–200.
[14] Y. Chen, Z. Shi, H. Li, W. Zhao, Y. Liu, and Y. Qiao, "HIMALIA: Recovering compiler optimization levels from binaries by deep learning," in *Proc. SAI Intell. Syst. Conf.* Cham, Switzerland: Springer, 2018, pp. 35–47.
[15] K. Cho, B. van Merriënboer, C. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1724–1734.
[16] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proc. USENIX Secur. Symp.*, 2017, pp. 99–116.
[17] J. Chung, C. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," in *Proc. NIPS Workshop Deep Learn.*, 2014, pp. 1–9.
[18] S. Datta, "DeepObfusCode: Source code obfuscation through sequence-to-sequence networks," *CoRR*, vol. abs/1909.01837, pp. 1–11, Sep. 2019. [Online]. Available: http://arxiv.org/abs/1909.01837
[19] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489.
[20] M. Gaudesi, A. Marcelli, E. Sanchez, G. Squillero, and A. Tonda, "Malware obfuscation through evolutionary packers," in *Proc. Companion Publication Annu. Conf. Genetic Evol. Comput.*, Jul. 2015, pp. 757–758.
[21] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "LEMNA: Explaining deep learning based security applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 364–379.
[22] Z. Kan, H. Wang, L. Wu, Y. Guo, and D. X. Luo, "Automated deobfuscation of Android native binary code," *CoRR*, vol. abs/1907.06828, pp. 1–14, Jul. 2019. [Online]. Available: http://arxiv.org/abs/1907.06828
[23] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proc. Workshop Binary Anal. Res.*, 2019, pp. 1–11.
[24] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. ICLR Workshop Poster*, 2013, pp. 1–12.
[25] Y. Otsubo, A. Otsuka, M. Mimura, T. Sakaki, and H. Ukegawa, "o-glassesX: Compiler provenance recovery with attention mechanism from a short code fragment," in *Proc. Workshop Binary Anal. Res.*, 2020, pp. 1–12.
[26] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2014, pp. 701–710.
[27] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, and M. Debbabi, "BinComp: A stratified approach to compiler provenance attribution," *Digit. Invest.*, vol. 14, pp. S146–S155, Aug. 2015.
[28] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2011, pp. 100–110.

[29] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Extracting compiler provenance from program binaries," in *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*, 2010, pp. 21–28.

[30] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Comput. Surveys*, vol. 46, no. 1, pp. 1–32, Oct. 2013.

[31] S. Schrittwieser and S. Katzenbeisser, "Code obfuscation against static and dynamic reverse engineering," in *Proc. Int. Workshop Inf. Hiding*. Berlin, Germany: Springer, 2011, pp. 270–284.

[32] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Comput. Surveys*, vol. 49, no. 1, pp. 1–37, Jul. 2016.

[33] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Network and Distribted Systems Security*. San Diego, CA, USA: Internet Society, 2008, pp. 1–13.

[34] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proc. USENIX Secur. Symp.*, 2015, pp. 611–626.

[35] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 3145–3153.

[36] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," in *Proc. 2nd Int. Conf. Learn. Represent. (ICLR)*, Banff, AB, Canada, 2014, pp. 1–8.

[37] Z. Tian, T. Liu, Q. Zheng, E. Zhuang, M. Fan, and Z. Yang, "Reviving sequential program birthmarking for multithreaded software plagiarism detection," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 491–511, May 2018.

[38] Z. Tian, Q. Zheng, T. Liu, M. Fan, E. Zhuang, and Z. Yang, "Software plagiarism detection with birthmarks based on dynamic key instruction sequences," *IEEE Trans. Softw. Eng.*, vol. 41, no. 12, pp. 1217–1235, Dec. 2015.

[39] A. H. Toderici and M. Stamp, "Chi-squared distance and metamorphic virus detection," *J. Comput. Virol. Hacking Techn.*, vol. 9, no. 1, pp. 1–14, Feb. 2013.

[40] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "Ensemble adversarial training: Attacks and defenses," 2018, *arXiv:1705.07204*. [Online]. Available: http://arxiv.org/abs/1705.07204

[41] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 659–673.

[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[43] P. Wang, Q. Bao, L. Wang, S. Wang, Z. Chen, T. Wei, and D. Wu, "Software protection on the go: A large-scale empirical study on mobile app obfuscation," in *Proc. 40th Int. Conf. Softw. Eng.*, May 2018, pp. 26–36.

[44] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 674–691.

[45] S. Yang, Z. Shi, G. Zhang, M. Li, Y. Ma, and L. Sun, "Understand code style: Efficient CNN-based compiler optimization recognition system," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2019, pp. 1–6.

[46] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, 2016, pp. 1480–1489.

[47] X. Zhang, R. Jiang, T. Wang, and J. Wang, "Recursive neural network for video deblurring," *IEEE Trans. Circuits Syst. Video Technol.*, early access, Nov. 3, 2020, doi: 10.1109/TCSVT.2020.3035722.

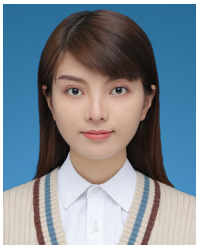
[48] G. Zhao and J. Huang, "DeepSim: Deep learning code functional similarity," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Oct. 2018, pp. 141–151.

[49] Y. Zhao, Z. Tang, G. Ye, D. Peng, D. Fang, X. Chen, and Z. Wang, "Semantics-aware obfuscation scheme prediction for binary," *Comput. Secur.*, vol. 99, pp. 1–20, Dec. 2020.

[50] G. Zheng, S. Mukherjee, X. L. Dong, and F. Li, "OpenTag: Open attribute value extraction from product profiles," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2018, pp. 1049–1058.

[51] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proc. Netw. Distrib. Syst. Secur. Symp.* San Diego, CA, USA: Internet Society, 2019, pp. 1–15.

ZHENZHOU TIAN was born in Shandong, China, in 1987. He received the B.S. and Ph.D. degrees in computer science and technology from Xi'an Jiaotong University, China, in 2010 and 2016, respectively. He is currently a Lecturer with the School of Computer Science and Technology, Xi'an University of Posts and Telecommunications. His research interests include software and system security, program similarity analysis, and software behavior analysis.



YAQIAN HUANG was born in Anhui, China, in 1997. She is currently a Master Student with the Xi'an University of Posts and Telecommunications. Her research interests include deep learning based program analysis and smart contract analysis.

BORUN XIE was born in Shaanxi, China, in 1996. He is currently a Master Student with the Xi'an University of Posts and Telecommunications. His research interests include binary code analysis and program provenance analysis.

YANPING CHEN was born in Shaanxi, China, in 1979. She received the Ph.D. degree in computer science and technology from Xi'an Jiaotong University, China, in 2007. She is currently a Professor with the School of Computer Science and Technology, Xi'an University of Posts and Telecommunications. Her research interests include software service computing, big data analysis, and cybersecurity.

LINGWEI CHEN received the Ph.D. degree in computer science from West Virginia University, in 2019. He is currently a Postdoctoral Scholar with the College of Information Sciences and Technology, The Pennsylvania State University. Prior to that, he was a Software Engineer with the Software Development Center, Agricultural Bank of China. He also got internship experience at Tencent and Yahoo! for research and development. His research interests include machine learning and cybersecurity.

DINGHAO WU (Member, IEEE) received the Ph.D. degree from Princeton University, in 2005. He is currently an Associate Professor with the College of Information Sciences and Technology, The Pennsylvania State University. Prior to joining Penn State, he was a Research Engineer with Microsoft in the Center for Software Excellence and the Windows Azure Division. His research interests include software systems, including software security, software analysis and verification, software engineering, and programming languages.