7-2023

BinAlign: Alignment Padding Based Compiler Provenance Recovery

MALIHA ISMAIL
Singapore Management University, malihai.2020@phdcs.smu.edu.sg

Yan LIN

DongGyun HAN

Debin GAO
Singapore Management University, dbgao@smu.edu.sg

BinAlign: Alignment Padding Based Compiler Provenance Recovery

Maliha Ismail[1(✉)], Yan Lin[2], DongGyun Han[3], and Debin Gao[1]

[1] Singapore Management University, Singapore, Singapore
{malihai.2020,dbgao}@smu.edu.sg
[2] College of Cyber Security, Jinan University, Guangzhou, China
yanlin@jnu.edu.cn
[3] Royal Holloway, University of London, London, UK
DongGyun.Han@rhul.ac.uk

Abstract. Compiler provenance is significant in investigating the source-level indicators of binary code, like development-environment, source compiler, and optimization settings. Not only does compiler provenance analysis have important security applications in malware and vulnerability analysis, but it is also very challenging to extract useful artifacts from binary when high-level language constructs are missing. Previous works applied machine-learning techniques to predict the source compiler of binaries. However, most of the work is done on the binaries compiled on Linux operating system. We highlight the importance and need to explore Windows compilers and the complicated binaries compiled on the latest versions of these compilers. Therefore, we construct a large dataset of real-world binaries compiled with four major compilers on Windows and four most common optimization settings. The complexity of the optimized programs leads us to identify specific patterns in the binaries that contribute to source compiler and specific optimization level. To address these observations, we propose an improved model based upon the state-of-the-art, and incorporate streamlined alignment padding features in the existing model. Thus, our improved model learns alignment instructions from binary code of portable executables and libraries using the attention mechanism. We conduct an extensive experimentation on a dataset of 296,169 unique and complex binary code generated from C/C++ applications. Our findings demonstrate that our proposed model significantly outperforms the state-of-the-art in accurately predicting the source compiler and optimization flag for complex compiled code.

Keywords: compiler provenance • alignment padding • Windows binaries • binary code similarity

1 Introduction

Microsoft Windows, which is currently the most dominant desktop operating system, commands a substantial market share.[1] However, it is also one of the most

[1] https://gs.statcounter.com/os-market-share/desktop/worldwide (Windows).

L. Simpson and M. A. Rezazadeh Baee (Eds.): ACISP 2023, LNCS 13915, pp. 609–629, 2023.
https://doi.org/10.1007/978-3-031-35486-1_26

targeted platforms for malicious activities by hackers and attackers. Statista[2] reports that 91% of newly developed ransomware in 2022 is intended to target Windows operating system. As a result, security analysts and researchers are interested to unleash all those methods that can aid in identifying the characteristics of binary code available in the wild.

When compiling a C/C++ source application, several flags are passed to the compiler, signaling the developer's intention to either keep or drop some information or to modify the original code in a more optimized version. The executable binary does not need to have knowledge of the compilation flags once it is compiled, as this information is no more required to execute the binary. However, these flags are useful during analysis to investigate whether a file was compiled with a specific flag that could expose vulnerabilities [7,32]. They are also useful, when compiler-optimization-specific security policies are applied on applications at the binary level for efficient CFI enforcement [17,18]. Thus, the security policies are applied varyingly for different compiler-optimization settings. Compiler provenance answers fundamental questions of malware analysis and software forensics, to know whether binaries are generated by similar toolchains [5,6,12,16,25,28–31]. It also aids the development of binary tools and analysis capabilities targeted at specific compilers or source languages [7,28,32]. Furthermore, the source compiler, version and development environment of binaries are amongst the most fundamental artifacts required for analysis at binary level.

Several studies have been conducted to develop techniques for compiler provenance analysis, which aim to identify source compilers and optimization flags [11,12,14,20,21,26,34,40]. However, the accuracy of these techniques may reduce due to the continuous developments in modern C/C++ optimizing compilers. Thus, with the advent of latest optimization strategies and newer Intel architectural extensions, the compiler provenance analysis approaches may become outdated [10,23]. While previous research into compiler provenance analysis techniques [5,25,28,30,31,33] demonstrate high accuracy and promising results, the study of compiler provenance on Windows platform is limited. Thus, we analyze the binaries compiled with Windows compilers and found significant patterns that could be a good indicator of the compiler and optimization levels.

Alignment padding [10,23] is a prominent feature of Windows binaries generated by modern compilers as it aligns the addresses for faster memory access and prevents processor faults. Since compilers optimize the code for either speed or size, their choice of instructions and preference for keeping data in the data section or in-line within the code determines the variation of alignment padding among compilers at different optimization settings. What makes the alignment padding interesting is the form and number of bytes emitted by the compiler with respect to the optimization [10,23]. Therefore, we identified four distinguishing patterns of alignment padding and integrate them into state-of-the-art model for learning and classifying the source compiler and optimization level.

[2] https://www.statista.com/statistics/701020/major-operating-systems-targeted-by-ransomware/.

With this observation, we propose BinAlign that leverages alignment padding in the binary code to help predict source compiler and optimization flags of the compiled code with improved accuracy. Thus, we train our proposed model to map the alignment padding in order to classify source compiler and optimization flag of the target binaries. The classification task identifies the toolchain used to generate unknown binaries and produces information that is required by binary analysis tools (i.e., for CFI enforcement and security patching) [17]. Thus, in order to evaluate the performance of o-glassesX [25] and our improved model, we mainly focus on answering the following research questions:

- RQ1. How effective is BinAlign in identifying the source compiler of a binary?
- RQ2. How effective is BinAlign in identifying two- (i.e., Od/O0 and Ox/O3) and four-level[3] (i.e., Od/O0, O1, O2 and Ox/O3) of compiler optimization settings?
- RQ3. How effective is BinAlign in identifying the joint source compiler and optimization settings?

Hence, we base our study on a set of 296,169 real-world binaries compiled with four major compilers, i.e., Microsoft Visual C++ (MSVC) [1], Clang-cl (Clangcl)[4], Intel C Compiler (ICC)[5], and Minimalist GNU Compiler Collection for Windows (MinGW) [2], in two- and four-level of varying compiler optimization settings. We achieved an overall f-measure (F1) of 0.978 when predicting the source compiler among four compilers and two compiler optimization levels. The main contributions of our paper are as follows:

- **An in-depth study on compiler provenance of binary code:** We investigate the characteristics of Windows binaries with the perspective of alignment bytes generated by modern compilers at four levels of compiler optimization settings.
- **Alignment padding based compiler provenance model:** We propose an improved convolutional neural network (CNN) that learns the special characteristics of alignment padding in the binaries and results in an improved performance.
- **Generality of our approach:** In order to evaluate the applicability of our method, we gather a collection of benign and malicious software binaries from real-world applications, compiled using different versions of the MSVC compiler. This helps us enhance the precision of our approach when analyzing complex and obfuscated malware binaries.

[3] For MSVC, Clangcl and ICC compilers, the optimization levels Od and Ox refer to *none* and *extreme* level of optimization. However, for the MinGW compiler, the corresponding optimization levels are O0 and O3, respectively.

[4] https://clang.llvm.org/docs/MSVCCompatibility.html (Clangcl with MSVC).

[5] https://support.alfasoft.com/hc/en-us/articles/360002874938 (Intel compatibility with MSVC).

Table 1. Alignment padding frequently found in common sections of the PE binary

Category	Section	Content	Paddings
Code section	`.text`	Executable code	`INT3`, `NOP`
Data Sections	`.data`	Read-write initialized data	`ZERO, DB`
	`.pdata`	Exception information	`ZERO, DB`
	`.rdata`	Read-only initialized data	`ZERO, DB`
	`.idata`	Import tables	none

2 Background and Related Work

2.1 Alignment Padding

In this section, we revisit the alignment padding generated by compilers in the binaries for aligning code and data. The term *Alignment padding* is referred to as the padding code placed as trailing sequence next to a control-flow transfer, or padding bytes at specific locations to align data and instructions in the binary (See Footnote 5). Alignment padding consists of an opcode and optionally an operand, similar to binary instructions on any architecture or platform [3].

Alignment Padding in PE Sections. Portable executable (PE) is a file format for executables, object code, and dynamic link libraries (DLLs) on Windows. Table 1 shows the sections commonly found in a PE binary and the types of alignment padding[6] that are most frequently found in each section. As described in Microsoft developer documentation [1], each section consists of different types of program data. The `.text` section contains executable code, while the data sections maintain data to execute the binary (i.e., `.data`, `.pdata`, `.rdata`, and `.idata`). Alignment padding is found in both the `.text` and data sections except for the `.idata` section, which contains the import directory and import address table (IAT). Table 2 shows the placement of four major types of alignment padding and filler instructions in the binary.

2.2 Machine Learning Approaches for Compiler Provenance

This section revisits the previous research conducted on compiler provenance and introduces the relevance of alignment padding for provenance recovery. Previous works utilized machine learning (ML) methods to perform compiler provenance recovery, since signature-based[7,8] methods depend on signatures database [9] to report the source compiler. On the other hand, ML-based methods learn the compiler-specific patterns and features to predict the source compiler of previously unseen binaries. Rosenblum et al. [30] were the first to extract syntactic and

[6] We use the instructions to represent alignment padding except `ZERO` as the type of alignment padding.
[7] https://github.com/horsicq/Detect-It-Easy.
[8] https://www.aldeid.com/wiki/PEiD.

Table 2. Types of alignment padding in the binary code

#	Align	Purpose	Usage
1	`NOP`	Program execution continues with the next instruction	Function-entry alignment in ICC and MinGW
2	`INT3`	Single-byte instruction for setting breakpoints for the debugger	Function-entry alignment in MSVC, Clangcl, ICC
3	`DB`	Reserve space for data	Data alignment in `.text` and data sections for MSVC, Clangcl, ICC, whereas data alignment in data sections only for MinGW
4	`ZERO`	`ADD` instruction with zero opcode and zero operand	Align code and data; may also update memory location, set carry, overflow, and zero flags
5	Filler	Pseudo NOPs	`MOV RAX,RAX`; `LEA RBX,[RBX+0]`; `XCHG RAX,RAX`

structural features from the binaries based on instruction idioms and graphlets. This work was followed by Chaki et al. [6], Xue et al. [39], and Rahimian et al. [27] that used various machine learning classifiers to identify similar chunks of code in the binary. Moreover, the past works [20,21,26,30,31,33,34] used binary-level control flow features and functions to predict program provenance. More recent works such as Ding et al. [8] proposed an Asm2Vec model for assembly code learning based on functions.

Although, past research acknowledged the alignment padding patterns in the binary [4,31,36,37], the significance of these patterns in relevance to program provenance has not been considered earlier. Rosenblum et al. [31] named the alignment bytes as gaps within the functions, whereas Andriesse et al. [4] considered these patterns as inline data within the code. Wang et al. [36,37] corroborated the reassembly of binary, while considering the memory alignment of data and function pointers. While considering all the past efforts, the compiler provenance recovery models [20,21,26,30,33,34] that take the control-flow features of the binary ignore the alignment padding that lie outside the control flow graph of the program such as function-entry and loop-entry alignment.

In this work, we chose o-glassesX [25] as our evaluation baseline for compiler provenance recovery leveraging alignment padding. This is because o-glassesX is state-of-the-art model with 97% accuracy, while utilizing short binary code from C/C++ object files. However, on recent compiler versions and a complex set of binaries, o-glassesX does not maintain to achieve the claimed accuracy. Therefore, we propose BinAlign to predict the source compiler and optimization level with better prediction accuracy.

2.3 o-glassesX Architecture

This section briefly explains the architecture of our baseline, o-glassesX. It uses natural language processing (NLP) techniques called the attention mechanism with convolutional neural network (CNN) to capture the characteristics of a single instruction by multiple local receptive fields [25]. The input unit of CNN is the local receptive field (i.e., kernel), whereas the output unit is the volume of kernel size (`K`), depth, and stride (`S`) length. The depth of the CNN refers to the number of filters, whereas stride is the step size of the kernel when traversing the width and height of the input binary image. In the preprocessing stage, o-glassesX disassembles binary into 128-bit fixed length instructions padded with zeros to construct a 2048-bit sequence of 16 instructions. The underlying architecture of neural network comprises the following CNN layers:

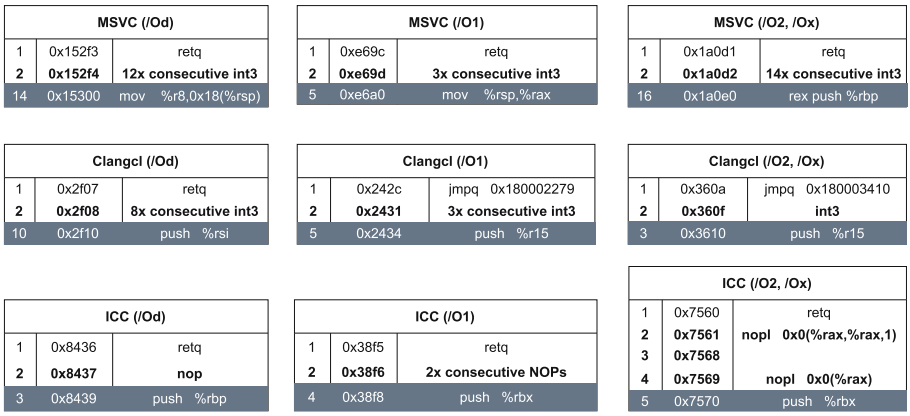
- The first layer takes 2048 bits of binary code as input. Each unit is a single-dimension 128-unit kernel, with stride length of 128 and depth of 96.
- The second layer is the positional encoding layer with 256 filters to a 16×96 input volume with a stride of 1. This layer captures the relationship between two adjacent instructions. The instructions in positional feed-forward network (PFFN) in Transformer are arranged into two dimensions by setting the stride and kernel size to 1.
- The third, fourth and fifth layers correspond to attention, batch normalization and fully connected layers with `K` nodes as output classes to classify the network, respectively. RELU is the activation function in each intermediate layer, whereas the final layer uses softmax for classification. In the model's back-propagation algorithm, the stochastic gradient descent (SGD) method is used to minimize the error function [25].

3 Motivation

Our motivation for this study is the observation of alignment paddings and their placement in the binary with respect to different compilers and optimization settings. To motivate the idea of leveraging alignment padding for compiler provenance recovery, we analyze the binary code of the function `sqlite3_str_vappendf` compiled with three compilers and optimization settings, and see if the corresponding alignment padding presents unique patterns. Listing 1 shows different snippets of the binary code disassembled at the entry of a variadic function in *sqlite3* C application.

```
SQLITE_API void sqlite3_str_vappendf(sqlite3_str *pAccum, const char
    *fmt, va_list ap) {...}
```

From Listing 1, our first observation is that MSVC and Clangcl compilers prefer to insert `INT3` bytes at the entry of a function, whereas ICC emits `NOP` in-place of `INT3` for the function-entry alignment. In this particular example, the alignment padding at the function-entry in O2 and Ox does not differentiate from



Listing 1 – Alignment padding at the entry of a variadic function *sqlite3_vtr_vappendf* in sqlite3 application.

each other. This is because the compilers will only generate different binary code in the corresponding optimization levels, when there exist duplicate copies of constant data elements and function definitions in the binary.[9] To demonstrate the frequency of alignment padding in binaries, we look at another example among the complex set of C projects in our database (i.e., *openssl*), as listed in Table 3. Here, we observe that the alignment padding in O2 and O3 vary in the MinGW compiler as compared to the other three compilers. This is because the MinGW compiler in O3 applies aggressive optimization strategies, like function inlining and loop unrolling, as compared to O2.

Moreover, the frequency of alignment in `.text` section highlights that `INT3` is the most frequently used alignment padding for the functions compiled with MSVC and Clangcl compilers. The reason is due to the fact that compilers emit `INT3` instruction as debugger trap to gracefully handle the execution in case of an exception (See Footnote 9). In contrast, MinGW compiler utilizes a larger number of `NOP`s for aligning the optimized instructions.

Interestingly, we found that `DB` and `ZERO` are frequently emitted alignment bytes in the data sections of compiled binaries at higher optimization levels. Therefore, to favor the small size of optimized binaries, compilers allocate data sections for alignment padding and emit reduced code in the `.text` section, respectively. Overall, all compilers emit frequent `ZERO` alignment padding in the data sections of the binaries, including the *end-of-section* alignment padding (See Footnote 9). Thus, we can say that there is a significant variation of alignment padding found in the binaries compiled with various compilers and at different optimization settings on Windows. Therefore, to achieve the best possible performance, it is a common practice for compilers to enforce natural alignment of both data and code [2,22]. However, the compiler's strategy for alignment

[9] https://docs.microsoft.com/en-us/cpp/build/reference/compiler-options-listed-alphabetically?view=msvc-160.

Table 3. Number of Alignment padding bytes in the `.text` and `.data` sections of *openssl* application, compiled with four compilers at four optimization levels.

Section		`.text`				`.data`			
Padding	Opt	MSVC	ICC	Clangcl	MinGW	MSVC	ICC	Clangcl	MinGW
`#NOP`	`Od/O0`	52	648	140	12,109	156	67	85	124
	`O1`	15	564	70	6,630	126	90	75	124
	`O2`	433	871	597	25,499	142	99	56	127
	`Ox/O3`	433	871	597	26,481	142	99	56	138
`#DB`	`Od/O0`	264	0	1,419	15	2,710	2,594	3,635	2,665
	`O1`	58	0	1,347	16	1,855	1,829	2,129	2,685
	`O2`	269	98	2,644	14	1,817	3,825	2,092	2,656
	`Ox/O3`	269	98	2,644	16	1,817	3,825	2,092	4,260
`#INT3`	`Od/O0`	3,438	117	15,146	0	47	48	55	64
	`O1`	671	144	12,156	0	49	66	50	66
	`O2`	2,608	188	14,125	0	58	53	62	51
	`Ox/O3`	2,608	188	14,125	0	58	53	62	69
`#ZERO`	`Od/O0`	164	0	2	9	11,726	15,388	14,710	17,477
	`O1`	150	0	6	9	13,646	16,422	13,322	17,492
	`O2`	222	9,682	4	9	12,454	18,714	11,120	17,283
	`Ox/O3`	222	9,682	4	9	12,454	18,714	11,120	16,263

padding varies from platform to platform. Thus, to compare Windows and Linux, Windows specify additional alignment options to align the sections and pages of PEs and DLLs on a 4K-byte boundary (See Footnote 9). Whereas, the sections on Linux are aligned on a 4-byte boundary [22,23]. These specifications are additional to the data alignment for optimized code constructs and transfer operations [22,35]. It is worth noting here that ELF x86-64 ABI does not require the virtual and physical address to be page-aligned. Though different from Linux, the alignment padding on Windows shows interesting patterns in the compiled binary, which is significant for compiler provenance. We thus highlight the importance of alignment padding on Windows and demonstrate that it is more useful for compiler provenance.

4 Compiler Provenance Recovery Model

In Sect. 3, we saw that different compilers and optimization levels emit unique signatures in the binary code compiled with different settings. In this section, we review the state-of-the-art deep learning model, o-glassesX and present our enhanced model, BinAlign. A deep neural network trained completely on data without domain knowledge might be non-explainable [38], whereas a system based entirely on expert knowledge may have limitations due to insufficient inference logic [28]. Xu et al. [38] state that adding node embeddings to control flow graph (CFG) of binary functions enhance the performance of the underlying binary similarity model. With this background, we improve the existing compiler provenance model, o-glassesX, and embed expert knowledge of alignment padding into the CNN based deep learning model.

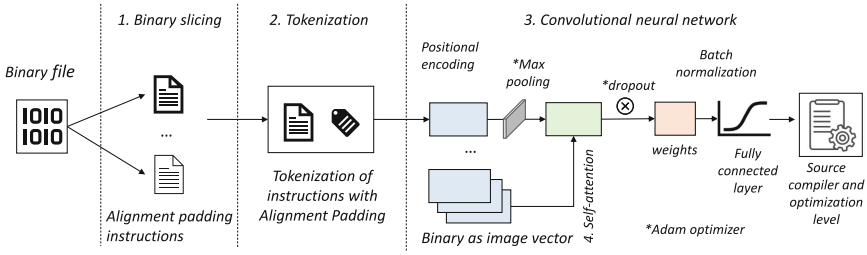


Fig. 1. Design overview of BinAlign and improved BinAlign architecture

4.1 BinAlign

The architecture of BinAlign is illustrated in Fig. 1. BinAlign follows the same approach as o-glassesX for classifying the binary instructions into different classes of compiler provenance. It is thus composed of the following three major components,[10] i.e., 1) binary slicing, 2) padding tokenization, 3) CNN. The neural network consists of the following core layers, i.e., i) positional encoding, ii) attention block, iii) batch normalization, and iv) fully connected layer.

The input to the network is a sequence of 128-bit fixed-length instructions that are embedded with alignment padding information. Thus, binary instructions are read in the form of image vector encodings. An image vector is a numerical representation of the binary in which each pixel is represented by a value of either 0 or 1.

To incorporate alignment padding in the underlying CNN architecture, we slice the binary code and detect patterns associated with alignment padding. We then categorize the alignment padding instructions into different groups and assign them the corresponding tokens (denoted by `TOK`), as shown at lines 6, 8, 10, and 12 of Algorithm 1. Thus, we encode the tokens along with the instructions and input the vectors into positional encoding layer preceding the attention layer of BinAlign. Following the approach in previous work [24], we pad all instructions with zeros to make them a uniform 16-byte length. Additionally, our approach considers all variations of the `NOP` instruction (as outlined in Table 4) to be incorporated into the model.

After tokenization, the positional encoding layer of BinAlign captures the relationship between two adjacent instructions. Thus, the positional encodings are learnt by the attention layer that utilizes self-attention to generate weights and focuses on a portion of the input information to classify binary sequences. Finally, the output of the attention block is passed through batch normalization layer to stabilize the network. Thus, the final layer of CNN (i.e., the fully connected layer) classifies the network into various output classes. RELU [25] is the activation function in each intermediate layer, whereas the final layer uses softmax for classification. In the model's back-propagation algorithm, the stochastic gradient descent (SGD) [25] method is used to minimize the error function.

[10] The improvements (i.e., additional layers and optimization algorithm) marked with * in Fig. 1, belong to the improved BinAlign design.

Algorithm 1. Binary Slicing and Instruction Tokenization

```
1: procedure FetchAlignmentPaddings(binary)
2:     foreach Insn in binaryCode
3:     if instruction == AlignmentPadding then
4:         Do foreach {Insn} in the AlignmentPadding
5:         if opcode_Insn == DATA then
6:             Assign TOK of "DB" instructions
7:         else if opcode_Insn == ZERO then
8:             Assign TOK of "ZERO" instructions
9:         else if opcode_Insn == INT3 then
10:            Assign TOK of "CC" instructions
11:        else if opcode_Insn == NOP then
12:            Assign TOK of "NOP" instructions
13:    else if instruction ∈ Filler then
14:        Assign TOK of "Filler" instructions
15:    else if instruction ∈ non-Alignment then
16:        Assign TOK of "non-padding" instruction
```

Table 4. Multi-byte NOP alignment padding

Opcode	Operand	no. of Bytes	Hex representation
NOP	<no operand>	1	90
NOP	<no operand>	2	6690
NOP	WORD [RAX+RAX+0x0]	9	660f1f840000000000
NOP	WORD [RAX+RAX+0x0]	10	662e0f1f840000000000
NOP	DWORD [RAX]	3	0f1f00
NOP	DWORD [RAX+0x0]	4	0f1f4000
NOP	DWORD [RAX+RAX+0x0]	5	0f1f440000

4.2 Improved BinAlign

Due to our added attributes in the form of alignment padding tokens (see Fig. 1) in the existing model [25], we need to enhance the underlying architecture with additional layers. This is because the alignment padding is not uniformly distributed in the binary. For example, at one point the compiler may align the end-of-section with a large number of recurring padding bytes, whereas at another location, a multi-byte instruction may be generated to enforce the instruction alignment. In Table 5, we present nine scenarios in which the compilers emit alignment padding at different locations in the binary. Therefore, it becomes necessary for BinAlign to include additional layers in the underlying architecture to enhance its performance. Thus, we add maxpooling, dropout and adam optimizer to enhance the basic architecture of BinAlign and name it as the improved BinAlign as shown in Fig. 1. We briefly explain the additional layers and optimization in this section.

Table 5. Nine scenarios when compilers emits alignment padding at different locations in the binary with respect to optimization

#	Placement in Binary	Purpose and Location	Impact of Optimization	Align type
1	Data interleaved in Code	inline data in .text section	increase with optimization	`DB`
2	Import Functions	before a branch instruction	decrease with optimization	`NOP`
3	Exception handling (EH) functions	aligned jump tables	less in MSVC & Clangcl, intense in ICC at O2, Ox[a]	`NOP`
4	Intrinsic functions	compiler's inlined functions	expansion of function vary with compiler family & optimization [19]	`NOP`, `DB`
5	Function-entry alignment	before subroutine or EH function	align code and data	`INT3`, `NOP`
6	Common Runtime (CRT) routines	handcoded assembly routines	intense use of CRT at higher optimization	`ZERO`, `DB`
7	End-of-segment alignment	PE sections are aligned	decrease with higher optimization	`ZERO`, `DB`
8	Vector operations	128-bit multimedia operands are aligned	`MMX` and `SSE (XMM)` instructions aligned at 16 Byte address	`NOP`, `DB`
9	Branch Alignment	align branch target to a multiple of 16	increase with optimization	`NOP`

[a] https://support.alfasoft.com/hc/en-us/articles/360002874938 (Intel compatibility with MSVC).

(1) *Pooling* [15] is a regularization technique that reduces overfitting, whereas *max pooling* decreases the computational cost by reducing the number of parameters to learn the features. We add a max-pooling layer after the convolution layer to extract shallow features from binary code with `K` as `96`, and stride length as `128`. Thus, pooling assists deep learning architectures to reduce computational cost caused by the dimensionality reduction problem [15].

(2) *Dropout* [15] assists the neural network in achieving high-quality performance on the test set and prevents the model from overfitting. We apply dropout to the fully connected layer of the neural network. Thus, by utilizing max-pooling and dropout, we aim to achieve the stochastic pooling in terms of activation picking, inspired by dropout regularization approach [15]. Here, we add a dropout layer with a rate of 0.5.

(3) *Adam optimization* [13] is an extension to classical stochastic gradient descent (SGD) to update network weights iteratively based on training data. SGD algorithm maintains a single learning rate (i.e., α) for weight updates which does not change during training. It has two more extensions, i.e., AdaGrad [41] and RMSProp [41] and Adam optimizer combines the benefits of both extensions of SGD. Thus, in order to optimize for better accuracy,

we replace SGD with Adam optimizer having following parameters, i.e., a) α as the coefficient for learning rate is set to 0.001, b) $\beta 1$ and $\beta 2$ as the exponential decay rates are set to 0.9 and 0.999, respectively, and c) ϵ is initialized as 1e−08. Thus, we train the improved BinAlign model over all the unoptimized and optimized binaries to learn the source compiler and optimization level classification. Similar to o-glassesX, we parse the binaries with distorm3[11] disassembler and get x86-64 assembly instructions.

5 Experimental Design

In this section, we explain our experimental design to evaluate the precision (P), recall (R) and f-measure (F1) of inferring the source compiler, and optimization level for o-glassesX, BinAlign (i.e., state-of-the-art with alignment padding), and improved BinAlign (i.e., state-of-the-art with alignment padding and additional layers in the CNN architecture). Hence, our evaluation is performed on a server machine having Ubuntu 22.04.1 LTS OS with 3.5 GHz and upto 64 CPU core processors with 132 GB RAM and 4 GeForce RTX2080 Ti GPUs having 12GB of memory. We thus perform experimentation on a large collection of binaries compiled with latest compilers on Windows, as they implement modern compiler optimization strategies. Here it is worth mentioning that we are not replicating the experiments in the o-glassesX paper. We therefore, focus on compiler provenance inference in x86-64 architecture for benign code and x86 for malicious code. The dataset is compiled from a set of 457 well-known C/C++ open-source projects using four commonly used compilers, i.e., MSVC-19, Clangcl-10, ICC-2021 and MinGW-10.

For the ground truth, we record the source compiler and optimization level label and compile all application in release build with debugging symbols enabled. Since, the debug information is present in separate PDB files that provide us information about the function-entry addresses, we study the differences in the alignment padding at the function start locations using the information extracted from the symbols. However, it is worth noting here that all tasks of provenance recovery training and inference are performed on the stripped binaries.

We therefore generate a diverse set of binaries that range in size from a few kilobytes to 40 megabytes. For each project, we build dependent libraries and programs separately. Further details regarding some of the projects in our dataset can be found in Table 6. Hence, we publish our compiled dataset and program source code for further study and research[12] (Table 7).

[11] https://pypi.org/project/distorm3.
[12] https://github.com/mali-arf/binalign_comp.

Table 6. The selected C/C++ projects from Github in our dataset

Project	Description
ogg-vorbis	audio encoder/decoder for lossy compression
cmake	a cross-platform, open-source build system generator
curl	library for data transfer with url syntax
doxygen	document generation tool from annotated C++ sources
eigen	C++ template library for linear algebra, matrices, vectors, numerical solvers, etc.
gflags	C++ library with command-line flags for strings, etc.
glm	C++ openGL Mathematics library for graphics
glog	C++ google logging (glog) module
libevent	a library that provides asynchronous event notifications
llvm	an open source compiler and toolchain
onednn	oneAPI deep neural network library optimized for Intel and Xe architectures
opencl	Open Computing Language for heterogeneous platforms like CPUs, GPUs, etc.
openssl	library to secure communications over computer networks against eavesdropping
Microsoft lib	C++ Standard Template Library (STL) for MSVC toolset and Visual Studio IDE
sqlite	a relational database management system contained in a C library
vtk	an image processing, 3D graphics, volume rendering, and visualization toolkit
zlib	data compression library used in gzip file compression programs

Table 7. The number of compiled binaries in our dataset

Compiler	Version	number of binaries			
		`Od/O0`	`O1`	`O2`	`Ox/O3`
MSVC	19.28.29	28,359	23,201	21,982	32,705
Clangcl	10.0.0	18,834	9,938	11,512	19,354
ICC	2021.1.2	14,475	10,193	11,816	14,798
MinGW	10.3.0	22,606	17,573	16,325	22,498
Total		84,274	60,905	61,635	89,355

We collect C/C++ applications in our dataset comprising of a large number of stars and widely used in binary analysis research. We, then construct a dataset of 296,169 binaries compiled with four major compiler optimization settings (see Sect. 5).

Thus, we fine-tune the evaluation models to get best results from o-glassesX, BinAlign and Improved BinAlign. o-glassesX utilizes four main hyperparameters that are tuned to achieve the best results in each model. Therefore, all models achieve the best average accuracy at the same hyperparameter setting with i) *batch size* as 1000, ii) *instruction length* as 64, iii) *sample size* as 100,000, and, iv) *epoch* as 50. Hence, we used 4-fold cross-validation to perform training and testing of data for four tasks, i.e., source compiler inference, compiler optimization inference, and joint source compiler with two and four levels of compiler optimization inference, respectively. The average results on the compiler provenance tasks are shown in Tables 8, 9, 10 and 11.

6 Results

In this section, we present our experimental results in the form of answers to the RQs discussed in Sect. 1. To ensure a fair comparison with previous research, we

calculate the average test results for each model in order to draw conclusions about the source compiler, optimization levels, and joint source compiler and optimization levels.

6.1 RQ1. How Effective is BinAlign in Identifying the Source Compiler of Binaries?

Table 8 measures our models' recall (R), precision (P) and f-measure (F1) while inferring the source compiler for the three models. Overall, our improved BinAlign model shows increased performance while inferring the source compiler of the test binaries. One of the reasons for improved performance on MSVC compiled binaries is the inference of `NOP` bytes to align the data for compiler-specific intrinsic functions. Which the model learns to correctly infer the compiler specific alignment. Similarly, for ICC compiled binaries, the improved BinAlign performs much better than o-glassesX for classifying the NOP bytes embedded before every `CALL` to the internal compiler functions.

6.2 RQ2. How Effective is BinAlign in Identifying the Compiler Optimization Level of a Binary?

Tables 9a and 9b present the results on the classification of the two- and four-level of compiler optimizations, respectively. The results of the two-level optimization inference task, as presented in Table 9a, demonstrate that the improved BinAlign model can accurately determine the optimization level of the most optimized binaries, with a level of precision that is almost as high as that for the unoptimized binaries. However, Table 9b shows that the least correctly inferred compiler optimization setting for BinAlign is O2. This is because of an optimization strategy that is followed by the compilers to maintain a single copy for the common data and functions in the binary (i.e., COMDAT (See Footnote 9)). The optimization confines the alignment padding, thereby affecting the inference of the highly optimized binaries.

However, the improved BinAlign demonstrates a relatively higher accuracy in inferring class labels for O1 and Ox optimized binaries. This can be attributed to the optimization strategy employed by compilers on Windows, which generates separate copies of aligned functions and data with multiple definitions at the Ox optimization level. Conversely, the code generated by the O1 optimization level remains consistent across all compilers. This consistency arises because compilers tend to reserve most of the alignment padding in the data sections, which favors the reduced size of the binary code. Overall, our evaluation results show that optimizations performed by compilers generate variant and complex code, making it difficult for the compiler prediction neural network based models to learn the consistent patterns. However, the improved BinAlign significantly improves the existing state-of-the-art model in recovering the compiler optimization inference of real-world binaries.

Table 8. The result of source compiler inference

Compiler	o-glassesX			BinAlign			Improved BinAlign		
	R	P	F1	R	P	F1	R	P	F1
MSVC	0.9729	0.9797	0.9763	0.9953	0.9934	0.9943	0.9938	0.9972	0.9955
Clangcl	0.9465	0.9470	0.9467	0.9787	0.9809	0.9798	0.9739	0.9890	0.9814
ICC	0.9662	0.9697	0.9679	0.9890	0.9907	0.9899	0.9944	0.9862	0.9903
MinGW	0.9216	0.8988	0.9100	0.9643	0.9613	0.9628	0.9812	0.9646	0.9728
Overall	0.9576	0.9576	0.9576	0.9850	0.9850	0.9850	0.9873	0.9873	0.9873

Table 9. The result of two and four levels of compiler optimization level inference

(a) Two-level compiler optimization inference (i.e., Od/O0 and Ox/O3)

	o-glassesX			BinAlign			Improved BinAlign		
Opt	R	P	F1	R	P	F1	R	P	F1
Od/O0	0.9783	0.9736	0.9764	0.9888	0.9896	0.9892	0.9914	0.9902	0.9908
Ox/O3	0.9696	0.9750	0.9723	0.9883	0.9874	0.9879	0.9890	0.9903	0.9897
Overall	0.9739	0.9743	0.9743	0.9886	0.9886	0.9886	0.9902	0.9902	0.9902

(b) Four-level compiler optimization inference (i.e., Od/O0, O1, O2, Ox/O3)

	o-glassesX			BinAlign			Improved BinAlign		
Opt	R	P	F1	R	P	F1	R	P	F1
Od/O0	0.9839	0.9838	0.9838	0.9873	0.9885	0.9879	0.9915	0.9874	0.9894
O1	0.7365	0.7275	0.7320	0.7313	0.8078	0.7676	0.8002	0.8375	0.8184
O2	0.6971	0.6011	0.6456	0.6316	0.6701	0.6503	0.8248	0.8217	0.8232
Ox/O3	0.6371	0.6276	0.6323	0.7198	0.8714	0.7884	0.7495	0.9083	0.8213
Overall	0.7678	0.7472	0.7562	0.7730	0.8298	0.7995	0.8318	0.8811	0.8547

Table 10. Inference of joint source compiler with two levels of compiler optimization

Compiler-opt	o-glassesX			BinAlign			Improved BinAlign		
	R	P	F1	R	P	F1	R	P	F1
MSVC-Od	0.9667	0.9643	0.9655	0.9886	0.9743	0.9814	0.9991	0.9925	0.9958
MSVC-Ox	0.9357	0.9535	0.9445	0.9651	0.9795	0.9722	0.9735	0.9808	0.9772
Clangcl-Od	0.9779	0.9744	0.9761	0.9885	0.9888	0.9886	0.9985	0.9998	0.9992
Clangcl-Ox	0.9088	0.9212	0.9150	0.9599	0.9508	0.9553	0.9508	0.9618	0.9563
ICC-Od	0.9831	0.9905	0.9868	0.9945	0.9971	0.9958	0.9932	0.9995	0.9963
ICC-Ox	0.9519	0.9423	0.9471	0.9777	0.9840	0.9808	0.9732	0.9815	0.9773
MinGW-O0	0.9711	0.9668	0.9690	0.9949	0.9868	0.9908	0.9993	0.9942	0.9967
MinGW-O3	0.8438	0.7975	0.8200	0.9085	0.9164	0.9124	0.9367	0.9194	0.9279
Overall	0.9515	0.9515	0.9515	0.9772	0.9772	0.9772	0.9781	0.9787	0.9785

Table 11. Inference of joint source compiler with four levels of compiler optimization

Compile-opt	o-glassesX			BinAlign			Improved BinAlign		
	R	P	F1	R	P	F1	R	P	F1
MSVC-Od	0.9715	0.9695	0.9705	0.9874	0.9718	0.9795	0.9826	0.9737	0.9781
MSVC-O1	0.7692	0.6615	0.7113	0.7732	0.7961	0.7845	0.8880	0.7986	0.8409
MSVC-O2	0.4995	0.2488	0.3322	0.5806	0.0845	0.1476	0.5223	0.3005	0.3816
MSVC-Ox	0.7109	0.8602	0.7784	0.7113	0.9127	0.7995	0.7458	0.8791	0.8070
Clangcl-Od	0.9810	0.9808	0.9809	0.9883	0.9850	0.9866	0.9932	0.9816	0.9874
Clangcl-O1	0.6215	0.4529	0.5240	0.7532	0.5485	0.6348	0.7721	0.6354	0.6971
Clangcl-O2	0.3373	0.0696	0.1154	0.6477	0.5503	0.5951	0.6515	0.7314	0.6892
Clangcl-Ox	0.7392	0.8892	0.8073	0.6126	0.7292	0.6658	0.7023	0.6501	0.6752
ICC-Od	0.9888	0.9929	0.9909	0.9950	0.9943	0.9947	0.9941	0.9979	0.9960
ICC-O1	0.7487	0.5993	0.6657	0.7614	0.7666	0.7640	0.8759	0.8062	0.8396
ICC-O2	0.3853	0.1187	0.1815	0.6586	0.0890	0.1568	0.4455	0.2701	0.3363
ICC-Ox	0.7158	0.8959	0.7958	0.7243	0.9367	0.8169	0.7468	0.8624	0.8005
MinGW-O0	0.9782	0.9764	0.9773	0.9833	0.9917	0.9875	0.9928	0.9868	0.9898
MinGW-O1	0.9466	0.9372	0.9419	0.9736	0.9464	0.9598	0.9640	0.9678	0.9659
MinGW-O2	0.6356	0.6345	0.6351	0.6123	0.7067	0.6561	0.7051	0.7541	0.7287
MinGW-O3	0.6363	0.6250	0.6306	0.6373	0.5674	0.6003	0.7715	0.6957	0.7316
Overall	0.7422	0.6979	0.7053	0.7869	0.7385	0.7357	0.8077	0.7806	0.7896

6.3 RQ3. How Effective is BinAlign in Identifying the Joint Source Compiler and Optimization Level of a Windows Binary?

Tables 10 and 11 show the performance of three compiler provenance models, while predicting the joint source compiler and optimization level of the compiled binaries with two and four levels of optimization, respectively. For the two-level joint source compiler and optimization level inference as shown in Table 10, our improved BinAlign performs the best on compiler provenance recovery of MSVC and ICC compiled binaries with better inference on unoptimized binaries as compared to the highly optimized ones.

Moreover, from the results we can see that the neural network models do not perform perfectly well, while inferring the joint source compiler and optimization level, specifically for MSVC and ICC compiled test binaries, at O2 optimization level. This is because, one of the challenges for deep learning model is the highly optimized code and the inter-procedural optimization in optimized binaries.

Furthermore, it is observed that neural network models perform relatively better when the binaries are compiled with either unoptimized settings or with the most-aggressive optimization settings using MSVC and ICC compilers. This may be attributed to the fact that modern C/C++ compilers provide support for advanced vector extension (AVX) architecture, which generates over-aligned instructions, thus leading to improved performance (See Footnote 5). Also, our

test set instances comprise of instructions operating on floating point data, for which the ICC compiler emits alignment padding in the form of `DB` bytes to align the instructions and data for vector operations.

On the other hand, for unoptimized binaries, the improved BinAlign model learns the alignment padding patterns comparatively well, as the instructions are padded with `DB 90` and `DB 0CC` in the `.text` section of the compiled binaries, whereas `DB 0` in the data sections, respectively. One of the significant characteristics of the deep learning CNN models is their higher performance and better accuracy for the zero-padded data bytes [15]. Hence, from our evaluation results of joint prediction of source compiler and optimization level, we conclude that the improved BinAlign recovers the compiler provenance of Clangcl, and MSVC compiled binaries with a relatively better score as compared to other compilers.

6.4 Malware Case Study

To illustrate the generality of our approach for compiler provenance inference of real-world malware, we compile 113 C/C++ source code of Win32 malware downloaded from theZoo[13] repository. Since the malware source uses the core Windows APIs which are incompatible with Clangcl, ICC and MinGW compilers, we therefore compile the projects on three different versions of MSVC compiler, i.e., VS2015, VS2017 and VS2019, at four different optimization levels. Thus, we train our models on 64-bit benign programs and 32-bit malicious programs and test the models on malicious binaries only. The benign programs belong to the same set of programs as mentioned in Sect. 5, however for the current evaluation we compile them with MSVC compiler and perform testing with three different versions of the MSVC compiler.

Tables 12 and 13 present the malware families and the distribution of our malware dataset, respectively. In our dataset, we gather seven different families of malware programs in C/C++ language as shown in Table 12. The malware programs are then compiled with three versions of MSVC compiler in four different optimization settings as shown in Table 13, with the most successful compilation in version 2019 that supports the most APIs. Table 14a shows the overall classification result of compiler version prediction and compiler optimization prediction of our malware dataset. Hence, our results demonstrate that the improved BinAlign outperforms other models in predicting the compiler version of malware binaries. This is because adversaries may introduce binary padding to add junk data in the code and change the on-disk representation of the binaries. Here, we acknowledge that despite the obfuscation implemented in a smaller set of malicious binaries as compared to a larger set of benign programs, our improved model is able to predict compiler version and optimization level with a promising score.

[13] https://github.com/ytisf/theZoo.

Table 12. The number and types of binaries belonging to different families of malware.

Family	Malware Type	#bins
Dexter	Point of Sales Trojan	1
Rovnix	Bootkit	1
Carberp	Botnet	36
BJWJ	Banking Trojan	6
Anti_Rapport	Banking Trojan	11
Trochilus	Remote Access Trojan	40
ZeroAccess	Rootkit	18
Total		113

Table 13. The number of malware binaries successfully compiled with three different versions of MSVC compiler and at four different optimization levels.

Opt	VS2019	VS2017	VS2015	Total
`Od`	17	8	8	33
`O1`	13	5	5	23
`O2`	21	10	4	35
`Ox`	14	4	4	22
Total	65	27	21	113

Table 14. Compiler, version and optimization level inference of malware and custom aligned binaries

(a) Compiler version and optimization level inference of malware binaries

	Version			Opt		
Metrics	o-glassesX	BinAlign	Improved BinAlign	o-glassesX	BinAlign	Improved BinAlign
P	0.8540	0.8713	0.9160	0.8718	0.8825	0.9287
R	0.8548	0.8729	0.9170	0.8740	0.8827	0.9295
F1	0.8542	0.8719	0.9162	0.8722	0.8822	0.9290

(b) Compiler and optimization level inference of custom-aligned binaries

	Compiler			Opt		
Metrics	o-glassesX	BinAlign	Improved BinAlign	o-glassesX	BinAlign	Improved BinAlign
P	0.7923	0.8280	0.8557	0.6892	0.7708	0.7950
R	0.7927	0.8294	0.8548	0.6729	0.7709	0.7920
F1	0.7925	0.8287	0.8552	0.6811	0.7709	0.7935

6.5 Custom Alignment Padding

Considering a scenario when software developers or malware authors intentionally modify the compiler's default alignment settings, we conduct a case study comprising the binaries that enforce custom alignment padding in the compiled binaries. For that, we perform an extensive study of the compiler options that support aligning data within sections, structures, data packing and section alignment. Hence, we found that MSVC (See Footnote 9), Clangcl and ICC support four major alignment settings, defined as, i) `ALIGN`, ii) `FILEALIGN`, iii) `Zc:alignedNew`, and iv) `Zp16`. These options correspond to the section alignment in linear address space, alignment of sections to the output file, alignment of dynamically-allocated data and the packing of structure member alignment, respectively. On the other hand, MinGW compiler supports alignment of c++17 data standard, aligning data for functions, labels, jumps, and loops, respectively. Here, the compiler options to achieve the respective alignment is `-faligned-new`, `-falign-functions`, `-falign-jumps`, `-falign-labels`, and `-falign-loops`,

respectively [2]. Thus, we train the models with custom alignment of 8192 bytes for all the alignment options discussed above.

Therefore, to evaluate our models on custom-aligned compiled binaries, we train the compiler provenance models with, i) default alignment compiler settings, and ii) custom-alignment padding on the same dataset as evaluated in Sect. 6.1 to 6.3. We then test our trained models with custom-aligned binary code to measure their evaluation performance. Therefore, we utilize 70% of our total compiled data for training, while 30% of the custom-aligned binaries for testing. Hence, our evaluation results show that the improved BinAlign infers the source compiler and compiler optimization level of custom-aligned compiled binaries with a fairly decent score as shown in Table 14b.

7 Limitation

We evaluate BinAlign only on the selected options of custom alignment padding provided by the compiler. However, the alignment options in MinGW compiler (for example) for aligning data in C++17 standard, functions, loops, jumps and labels have a long range from 2^2 to 2^{16}. Hence, it would be interesting to assign labels to the differently padded binary code with varying alignment padding settings. We thus leave this as the future work. Additionally, this work can further be extended to incorporate the architecture-specific alignment padding [2].

8 Conclusion

In this work, we evaluate the state-of-the-art compiler provenance recovery model, o-glassesX and propose BinAlign that incorporates compiler-optimization-specific domain knowledge into the existing model. Thus, by learning the alignment padding in the binary code, our model infers the source compiler and optimization level over a diverse set of real-world binaries including 64-bit benign and 32-bit malware binaries. We thus believe that newer patterns of instructions for alignment padding may be explored to enhance the compiler provenance analysis. Moreover, this work can be extended to incorporate additional filler instructions to enhance the performance of compiler provenance analysis in optimized binaries.

Finally, we sincerely thank the authors of o-glassesX (Otsubo et al.) for providing us with the replication package along with the necessary guidance.

References

1. Microsoft Visual C++. https://visualstudio.microsoft.com/vs/features/cplusplus/. Accessed 23 Apr 2023
2. MinGW-w64 compiler (2023). https://www.mingw-w64.org/. Accessed 20 Apr 2023

3. Andriesse, D.: Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly (2018). https://books.google.com.sg/books?id=laWgswEACAAJ
4. Andriesse, D., Chen, X., van der Veen, V., Slowinska, A., Bos, H.: An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: 25th USENIX Security Symposium (2016)
5. Benoit, T., Marion, J.Y., Bardin, S.: Binary level toolchain provenance identification with graph neural networks. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (2021)
6. Chaki, S., Cohen, C., Gurfinkel, A.: Supervised learning for provenance-similarity of binaries. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 15–23 (2011)
7. Cifuentes, C., Gough, K.J.: Decompilation of binary programs. Softw. Pract. Experience **25**(7), 811–829 (1995)
8. Ding, S.H., Fung, B.C., Charland, P.: Asm2Vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 472–489. IEEE (2019)
9. Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. No Starch Press, USA (2011)
10. Grune, D., Van Reeuwijk, K., Bal, H.E., Jacobs, C.J., Langendoen, K.: Modern Compiler Design. Springer, Heidelberg (2012)
11. Ji, Y., Cui, L., Huang, H.H.: BugGraph: differentiating source-binary code similarity with graph triplet-loss network, pp. 702–715. ACM, New York (2021)
12. Ji, Y., Cui, L., Huang, H.H.: VESTIGE: identifying binary code provenance for vulnerability detection. In: Sako, K., Tippenhauer, N.O. (eds.) ACNS 2021. LNCS, vol. 12727, pp. 287–310. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78375-4_12
13. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
14. Koo, H., Park, S., Choi, D., Kim, T.: Semantic-aware binary code representation with BERT. arXiv preprint arXiv:2106.05478 (2021)
15. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. Commun. ACM **60**(6), 84–90 (2017)
16. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 207–226. Springer, Heidelberg (2006). https://doi.org/10.1007/11663812_11
17. Lin, Y., Gao, D.: When function signature recovery meets compiler optimization. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 36–52. IEEE (2021)
18. Lin, Y., Gao, D., Lo, D.: Resil: revivifying function signature inference using deep learning with domain-specific knowledge. In: Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy, pp. 107–118 (2022)
19. Lopes, B.C., Auler, R.: LLVM: Building a Modern Compiler Infrastructure (2020)
20. Massarelli, L., Di Luna, G.A., Petroni, F., Querzoni, L., Baldoni, R.: Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In: Proceedings of the 2nd Workshop on Binary Analysis Research (BAR) (2019)
21. Massarelli, L., Di Luna, G.A., Petroni, F., Baldoni, R., Querzoni, L.: SAFE: self-attentive function embeddings for binary similarity. In: Perdisci, R., Maurice, C.,

Giacinto, G., Almgren, M. (eds.) DIMVA 2019. LNCS, vol. 11543, pp. 309–329. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22038-9_15
22. Mitchell, M., Oldham, J., Samuel, A.: Advanced Linux Programming. New Riders, Berkeley (2001)
23. Muchnick, S., et al.: Advanced Compiler Design Implementation. Morgan Kaufmann (1997)
24. Otsubo, Y., Otsuka, A., Mimura, M., Sakaki, T.: o-glasses: visualizing x86 code from binary using a 1D-CNN. IEEE Access **8**, 31753–31763 (2020)
25. Otsubo, Y., Otsuka, A., Mimura, M., Sakaki, T., Ukegawa, H.: o-glassesx: compiler provenance recovery with attention mechanism from a short code fragment. In: Proceedings of the 3rd Workshop on Binary Analysis Research (2020)
26. Pizzolotto, D., Inoue, K.: Identifying compiler and optimization options from binary code using deep learning approaches. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME) (2020)
27. Rahimian, A., Nouh, L., Mouheb, D., Huang, H.: Binary code fingerprinting for cybersecurity
28. Rahimian, A., Shirani, P., Alrbaee, S., Wang, L., Debbabi, M.: Bincomp: A stratified approach to compiler provenance attribution. In: Digital Investigation, the Proceedings of the Fifteenth Annual DFRWS Conference (2015)
29. Ramshaw, M.J.: Establishing malware attribution and binary provenance using multicompilation techniques (2017). https://www.osti.gov/biblio/1390004
30. Rosenblum, N., Miller, B.P., Zhu, X.: Recovering the toolchain provenance of binary code. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA 2011 (2011)
31. Rosenblum, N.E., Miller, B.P., Zhu, X.: Extracting compiler provenance from program binaries. In: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE 2010 (2010)
32. Rosenblum, N.E., Zhu, X., Miller, B.P., Hunt, K.: Learning to analyze binary computer code. In: AAAI, pp. 798–804 (2008)
33. Shirani, P., Wang, L., Debbabi, M.: Binshape: scalable and robust binary library function identification using function shape. In: Detection of Intrusions and Malware, and Vulnerability Assessment (2017)
34. Tian, Z., Huang, Y., Xie, B., Chen, Y., Chen, L., Wu, D.: Fine-grained compiler identification with sequence-oriented neural modeling. IEEE Access **9**, 49160–49175 (2021)
35. TIS Committee: Executable and Linking Format (ELF) Specification (1995). https://refspecs.linuxfoundation/elf/elf.pdf
36. Wang, R., et al.: RAMBLR: making reassembly great again. In: NDSS (2017)
37. Wang, S., Wang, P., Wu, D.: Reassembleable disassembling. In: 24th USENIX Security Symposium, Washington D.C., pp. 627–642 (2015)
38. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC, pp. 363–376 (2017)
39. Xue, H., Sun, S., Venkataramani, G., Lan, T.: Machine learning-based analysis of program binaries: a comprehensive study. IEEE Access **7**, 65889–65912 (2019)
40. Yang, S., Shi, Z., Zhang, G., Li, M., Ma, Y., Sun, L.: Understand code style: efficient CNN-based compiler optimization recognition system. In: 2019 IEEE International Conference on Communications (ICC), ICC 2019 (2019)
41. Zou, F., Shen, L., Jie, Z., Zhang, W., Liu, W.: A sufficient condition for convergences of Adam and RMSProp. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 11127–11135 (2019)