BinAlign: Alignment Padding based Compiler Provenance Recovery

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 program 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, o-glassesX, 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. We show that our improved model surpasses state-of-the-art by a large margin in predicting source compiler and optimization flag of complex compiled code.

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

1 Introduction

Microsoft Windows, the most popular desktop operating system, holds a huge percentage of the market share [7]. Not only that, it is one of the most attractive platforms for hackers and attackers to launch malicious activities. According to Statista [6], 91% of newly developed ransomware in 2022 intends to attack Windows operating system. Therefore, researchers and security analysts are interested to unleash all those methods that can help identify characteristics of the 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 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 [13,41]. They are also useful, when compiler-optimization-specific security policies are applied on applications at the binary level for efficient CFI enforcement [24,25]. 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 [11,12,19,23,34,37–40]. It also aids the development of binary tools and analysis capabilities targeted at specific compilers or source languages [13, 37, 41]. Furthermore, the source compiler, version and development environment of binaries are amongst the most fundamental artifacts required for analysis at binary level.

There are various studies involved in the development of compiler provenance analysis techniques for the identification of source compilers and optimization flags [18,19,21,27,28,35,43,49]. However, the accuracy of these techniques may reduce with the evolving developments in the 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 [16,32]. Certainly, past researches in compiler provenance analysis techniques [11,34,37,39,40,42] show promising results with high accuracy, however, the study of compiler provenance on Windows platform is limited. Thus, we analyzed the executables and libraries compiled with Windows compilers and found significant patterns in the binary that could be a good indicator of the compiler and optimization levels.

Alignment padding [16,32] is a prominent feature of Windows binaries generated by modern compilers to align the addresses for faster memory access and to avoid processor fault. Since compilers optimize the code for speed or size, the choice of instructions and the preference of keeping data in the data section or in-line within code is a decision that results in alignment padding to vary from compiler-to-compiler at different optimization settings. The interesting thing about alignment padding is the form and the number of bytes emitted by the compiler with respect to the optimization [16,32]. We, therefore, indicate 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.

With this observation, we propose BinAlign, that leverages alignment paddings in the binary code to help predict source compiler and optimization flags of the compiled code with improved accuracy. We train our proposed model to map the alignment paddings in order to classify the source compiler and optimization flag of the target binaries. This 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) [24].

In order to evaluate the performance of o-glassesX 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-levels[1] (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) [30], Clang-cl (Clangcl) [2], Intel C Compiler (ICC) [1], and Minimalist GNU Compiler Collection for Windows (MinGW) [4], in two- and four-levels of varying compiler optimization settings. We achieved an overall f-measure 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 paddings in the binaries and results in an improved performance.
- **Generality of our approach**: To test the generality of our approach, we compile a set of real-world benign and malicious binaries on various versions of the MSVC compiler to achieve improved accuracy on the obfuscated malware binaries.

The remainder of this paper is structured as follows: Section 2 briefs the background of alignment padding in the binary code, and related work on compiler provenance. Section 3 explains our motivation to study the alignment patterns. Section 4 explains the state-of-the-art and our improved model. Section 5 details the experimental design and dataset. Section 6 explains our evaluation results and findings. Section 7 and Section 8 conclude our work with future directions.

2 Background and Related Work

2.1 Alignment Paddings

In this section, we revisit the alignment paddings generated by compilers in the binaries for aligning data and code. *Alignment padding* is referred to as the padding code placed as trailing sequence next to a control-flow transfer, or padding bytes to align data and instructions in the binary [1]. Alignment padding

[1] For MSVC, Clangcl and ICC compilers, Od and Ox refer to *none* and *extreme* level of optimization, whereas for MinGW it is O0 and O3, respectively.

Table 1 – Alignment paddings 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

Table 2 – The usage of each type of alignment padding with respect to their placement 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`

consists of an opcode and optionally an operand, similar to binary instructions on any architecture or platform [9].

Alignment Padding in PE Sections. Portable Executable (PE) is a file format for executables, object code, and dynamic link libraries on Windows. Table 1 shows the sections commonly found in a PE binary and the types of alignment paddings[2] that are most frequently found in each section. As described in Microsoft developer documentation [30], each section comprises 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 paddings are 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 paddings and filler instructions in the binary.

[2] We used the names of instructions to represent alignment paddings except `ZERO` as the type of alignment padding.

2.2 Machine Learning Approaches for Compiler Provenance

This section revisits the previous research conducted on compiler provenance and introduces the relevance of alignment paddings for provenance recovery. Previous works utilized machine learning (ML) methods to perform compiler provenance recovery, since signature-based methods depend on signatures database [3, 5, 15] 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. [39] were the first to extract syntactic and structural features from the binaries based on instruction idioms and graphlets. This work was followed by Chaki et al. [12], Xue et al. [48], and Rahimian et al. [36] that used various machine learning classifiers to identify similar chunks of code in the binary. Moreover, the past works [27,28,35,39,40,42,43] used binary-level control flow features and functions to predict program provenance. More recent works such as Ding et al. [14] proposed an Asm2Vec model for assembly code learning based on functions.

Although, past research acknowledged the alignment padding patterns in the binary [10, 40, 45, 46], the significance of these patterns in relevance to program provenance has not been considered earlier. Rosenblum et al. [40] named the alignment bytes as gaps within the functions, whereas Andriesse et al. [10] considered these patterns as inline data within the code. Wang et al. [45, 46] 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 [27, 28, 35, 39, 42, 43] that take the control-flow features of the binary ignore the alignment paddings 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 as our evaluation baseline to recover the compiler provenance, leveraging alignment paddings. 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 more complex set of hand-crafted 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.

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 paddings for compiler provenance recovery, we analyze the binary code of the following function 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, `sqlite3_str_vappendf` in *sqlite3* C application.

Clangcl (/Od)

1	0x2f07	retq
2	**0x2f08**	**int3**
3	**0x2f09**	**int3**
4	**0x2f0a**	**int3**
5	**0x2f0b**	**int3**
6	**0x2f0c**	**int3**
7	**0x2f0d**	**int3**
8	**0x2f0e**	**int3**
9	**0x2f0f**	**int3**
10	0x2f10	push %rsi

Clangcl (/O1)

1	0x242c	jmpq 0x180002279
2	**0x2431**	**int3**
3	**0x2432**	**int3**
4	**0x2433**	**int3**
5	0x2434	push %r15
6	0x2436	push %r14
7	0x2438	push %r13
8	0x243a	push %r12
9	0x243c	push %rsi
10	0x243d	push %rdi

Clangcl (/O2, /Ox)

1	0x360a	jmpq 0x180003410
2	**0x360f**	**int3**
3	0x3610	push %r15
4	0x3612	push %r14
5	0x3614	push %r13
6	0x3616	push %r12
7	0x3618	push %rsi
8	0x3619	push %rdi
9	0x361a	push %rbp
10	0x361b	push %rbx

MSVC (/Od)

1	0x152f3	retq
2	**0x152f4**	**int3**
3	**0x152f5**	**int3**
4	**0x152f6**	**int3**
5	**0x152f7**	**int3**
6	**0x152f8**	**int3**
7	**0x152f9**	**int3**
8	**0x152fa**	**int3**
9	**0x152fb**	**int3**
10	**0x152fc**	**int3**
11	**0x152fd**	**int3**
12	**0x152fe**	**int3**
13	**0x152ff**	**int3**
14	0x15300	mov %r8,0x18(%rsp)
15	0x15305	mov %rdx,0x10(%rsp)
16	0x1530a	mov %rcx,0x8(%rsp)

MSVC (/O1)

1	0xe69c	retq
2	**0xe69d**	**int3**
3	**0xe69e**	**int3**
4	**0xe69f**	**int3**
5	0xe6a0	mov %rsp,%rax
6	0xe6a3	mov %rbx,0x20(%rax)
7	0xe6a7	push %rbp
8	0xe6a8	push %rsi
9	0xe6a9	push %rdi
10	0xe6aa	push %r12
11	0xe6ac	push %r13
12	0xe6ae	push %r14
13	0xe6b0	push %r15
14	0xe6b2	lea -0x58(%rax),%rbp
15	0xe6b6	sub $0x120,%rsp
16	0xe6bd	movaps %xmm6,-0x48(%rax)

MSVC (/O2, /Ox)

1	0x1a0d1	retq
2	**0x1a0d2**	**int3**
3	**0x1a0d3**	**int3**
4	**0x1a0d4**	**int3**
5	**0x1a0d5**	**int3**
6	**0x1a0d6**	**int3**
7	**0x1a0d7**	**int3**
8	**0x1a0d8**	**int3**
9	**0x1a0d9**	**int3**
10	**0x1a0da**	**int3**
11	**0x1a0db**	**int3**
12	**0x1a0dc**	**int3**
13	**0x1a0dd**	**int3**
14	**0x1a0de**	**int3**
15	**0x1a0df**	**int3**
16	0x1a0e0	rex push %rbp

ICC (/Od)

1	0x8436	retq
2	**0x8437**	**nop**
3	0x8439	push %rbp
4	0x8439	sub $0x4d0,%rsp
5	0x8439	lea 0x20(%rsp),%rbp

ICC (/O1)

1	0x38f5	retq
2	**0x38f6**	**nop**
3	**0x38f7**	**nop**
4	0x38f8	push %rbx
5	0x38f9	push %rsi

ICC (/O2, /Ox)

1	0x7560	retq
2	**0x7561**	**nopl 0x0(%rax,%rax,1)**
3	**0x7568**	
4	**0x7569**	**nopl 0x0(%rax)**
5	0x7570	push %rbx

Listing 1 – Alignment padding at the entry of a variadic function *sqlite3_vtr_vappendf* in sqlite3 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 each other. This is because the compilers will generate different binary code in the corresponding optimization levels, when there exist duplicate copies of constant data elements and function definitions in the binary [29].

To demonstrate the frequency of alignment paddings in binary executables and dlls, we look at another example, which is one of the compiled applications 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

Table 3 – Number of Alignment paddings in the `.text` and `.data` sections of *openssl* application, compiled with four compilers at four different 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

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 for this is that these compilers emit `INT3` instruction as debugger trap to gracefully handle the execution in case of any exception [29]. Moreover, MinGW emits the most `NOP`s for the alignment of optimized instructions. Interestingly, we found that `DB` and `ZERO` are frequently generated alignment bytes in the data sections at high optimization levels of compiled binaries. Whereas 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 paddings in the data sections of the binaries, including the end-of-section alignment padding [29]. Thus, we can say that there is a significant variation of alignment paddings found in binaries compiled with various compilers and at varying optimization settings on Windows.

Generally, on all platforms compilers enforce data and code to be naturally aligned for optimal performance [2, 4, 17, 29, 31]. However the compiler's strategy for alignment padding varies from platform to platform. To compare Windows and Linux, Windows specify additional alignment options to align the sections and pages of portable executables and dlls on 4K byte boundary [29]. Whereas, the sections on Linux are aligned on 4-byte boundary [31, 32]. These specifications are additional to the data alignment for optimized code constructs and transfer operations [31, 44]. Moreover, the 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 emphasize that alignment padding on Windows is more useful for compiler provenance.

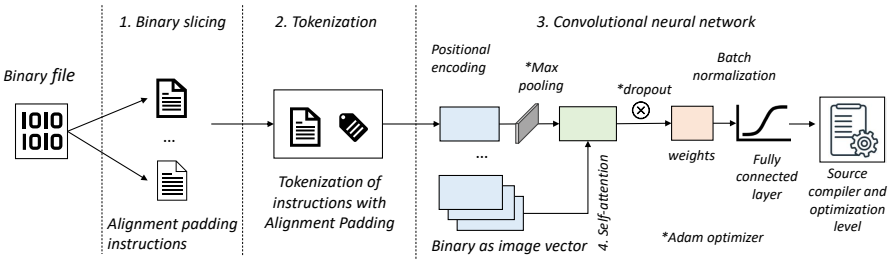

Fig. 1 – Design overview of BinAlign and improved BinAlign architecture

4 Compiler Provenance Recovery Model

In Section 3, we saw that different compilers and optimization levels emit unique signatures in the binary code compiled with different settings. In this section, we will 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 [47], whereas a system based entirely on expert knowledge may have limitations due to insufficient inference logic [37]. Xu et al. state that by adding node embeddings to control flow graph (CFG) of binary functions, resultantly enhances the performance of the underlying binary similarity model [47]. With this background, we improve the existing compiler provenance model, o-glassesX, and embed expert knowledge of alignment paddings into the CNN based deep learning model. A brief description of o-glassesX architecture is given in Appendix.

4.1 BinAlign

The architecture of BinAlign is illustrated in Figure 1. BinAlign follows the same approach as o-glassesX for classifying the binary instructions into different classes of compiler provenance. It comprises of the following three major components,[3] i.e., 1) binary slicing, 2) padding tokenization, 3) CNN. The neural network comprises of the following core layers, i.e., i) positional encoding (PE), 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. These instructions are embedded with alignment padding information. To embed alignment padding in the underlying CNN architecture, we first slice the binary code and identify the patterns related to alignment paddings. Algorithm 1 shows the step-by-step working of tokenization process. We assign the tokens (`TOK`) to particular category of alignment padding instructions as shown in line no. 6, 8, 10, and 12 of Algorithm 1. These tokens are encoded with the instructions and

[3] The improvements (i.e., additional layers and optimization algorithm) marked with * in Figure 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 – Different variations of the Multi-byte NOP alignment paddings are listed. The opcode of all the variations is NOP, whereas the operand varies depending upon the length of instruction.

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

input as vectors to the PE preceding the attention layer, as shown in Figure 1. Similar to previous work [33], all the instructions are padded with zeros to form a unified length of 16-byte instruction each. With our approach, we incorporate all of the variations of NOP (see Table 4) into our model.

The PE layer of BinAlign captures the relationship between two adjacent instructions. 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. The final layer of CNN (i.e., the fully connected layer) classifies the network into various output classes. RELU [34] 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) [34] method is used to minimize the error function.

4.2 Improved BinAlign

Due to added attributes in the form of alignment padding tokens (see Figure 1) in the core architecture of state-of-the-art model [34], we need to enhance the underlying architecture with additional layers. This is because the alignment

padding bytes are 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. The Appendix presents different locations in the binary where the compiler emits alignment padding instructions to enforce data and code alignment. 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 Figure 1. We briefly explain the additional layers and optimization in this section.

(1) *Pooling* [22] is a regularization technique to reduce overfitting and *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`. Pooling assists deep learning architectures to reduce computational cost caused by the dimensionality reduction problem [22].

(2) *Dropout* [22] also 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. By utilizing max-pooling and dropout, we aim to achieve the stochastic pooling in terms of activation picking, inspired by dropout regularization approach [22]. Thus, we add a dropout layer with a rate of 0.5.

(3) *Adam optimization* [20] 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 [50] and RMSProp [50]. Thus, Adam optimizer combines the benefits of both extensions of SGD. In order to optimize for better accuracy, we replace SGD with Adam optimizer with the 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 over all the unoptimized and optimized binaries to learn the source compiler and optimization level classification. Similar to o-glassesX, we parsed the binaries with distorm3[4] disassembler and get x86-64 assembly instructions.

5 Experimental Design

In this section, we explain our experimental design to evaluate the precision, recall and f-measure of inferring the source compiler, and optimization level for o-glassesX, BinAlign (i.e., the state-of-the-art with alignment padding), and improved BinAlign (i.e., the state-of-the-art with alignment padding and additional layers in the CNN architecture). Our evaluation is performed on a server

[4] https://pypi.org/project/distorm3

Table 5 – The number of compiled binaries and linked executables and dlls in our dataset

Compiler	Ver.	number of executables and dlls				number of binaries			
		Od/O0	O1	O2	Ox/O3	Od/O0	O1	O2	Ox/O3
MSVC	19.28.29	2,792	2,813	2,879	2,810	28,359	23,201	21,982	32,705
Clangcl	10.0.0	2,446	2,421	2,430	2,389	18,834	9,938	11,512	19,354
ICC	2021.1.2	2,275	2,281	2,331	2,345	14,475	10,193	11,816	14,798
MinGW	10.3.0	2,296	2,298	2,210	2,212	22,606	17,573	16,325	22,498
Total		9,809	9,813	9,850	9,756	84,274	60,905	61,635	89,355

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 perform evaluation on a large collection of binaries compiled on latest Windows compilers, as they implement the latest compiler optimization strategies. Note that this also means that we are not replicating the experiments in the o-glassesX paper. We 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.

The C/C++ applications in our dataset comprise of the highest number of stars in the github repository[5]. We ensure that these applications are widely used in binary analysis research. Our resulting dataset comprises of 296,169 different binaries compiled with four major compiler optimization settings (see Table 5). We record the source compiler and the optimization level as the ground truth label with which we compile the applications. We compile all the application programs and generate release builds with debugging symbols enabled. These symbols provide us information about the function-entry addresses in the resulting binary code. We study the differences in the alignment padding at the function start locations using the information extracted from symbols. We generate a variety of executables and dynamic link libraries whose sizes range from 12KB to 93MB. For each project, the dependent libraries and programs are built separately. The Appendix details the descriptions of some of the projects in our dataset. We publish our compiled dataset and program source code for further study and research[6].

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-level compiler optimization inference, respectively. The average results are shown in Tables 6 to 10.

5.1 Research Questions

We formulate three research questions to evaluate the effectiveness of the three models, i.e., i) o-glassesX, ii) BinAlign, and iii) Improved BinAlign.

[5] https://github.com/

[6] https://anonymous.4open.science/r/CompProv_Feb2023/

RQ1. How effective is BinAlign in identifying the source compiler of binary? This RQ aims to evaluate the effectiveness of BinAlign in determining the source compiler (i.e., MSVC, Clangcl, ICC, and MinGW) used to compile the programs.
RQ2. How effective is BinAlign in identifying the two-level and four-level optimized binary code in the dataset? This RQ measures the effectiveness of BinAlign in classifying two- and four levels of compiler optimizations, respectively.
RQ3. How effective is BinAlign in identifying the joint source compiler and optimization settings? This RQ measures the effectiveness of BinAlign to classify 8 classes for four compilers with two-optimization levels and 16 classes for four compilers and four-compiler optimization settings, respectively.

5.2 Model Fine-Tuning

We fine-tune the evaluation models to get the best results from state-of-the-art o-glassesX, BinAlign and Improved BinAlign. o-glassesX utilizes four main hyperparameters that are tuned to achieve the best results in each model. 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.

6 Results

In this section, we present our experimental results in the form of answers to the RQs presented in Section 5.1. For a fair comparison of the results with the earlier work, we used average values of the test results for each model to get the inference for 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 6 measures our models' recall (R), precision (P) and f-measure (F) 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. 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.

Table 6 – The result of source compiler prediction.

	o-glassesX			BinAlign			Improved BinAlign		
Class label	R	F	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 7 – The result of two-levels of compiler optimization level prediction. The two levels refer to none optimization (Od/O0) and maximum optimization (Ox/O3) as mentioned in o-glassesX, respectively.

	o-glassesX			BinAlign			Improved BinAlign		
Class label	R	P	F1	R	P	F1	R	P	F1
Od	0.9783	0.9736	0.9764	0.9888	0.9896	0.9892	0.9914	0.9902	0.9908
Ox	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

6.2 RQ2. How effective is BinAlign in identifying the Compiler Optimization level of a binary?

Tables 7 and 8 present the results on the classification of two- and four-level of compiler optimizations, respectively. For the two-level optimization inference task, we can see that in Table 7 the improved BinAlign model is able to infer the optimization level of most optimized binaries with almost the same precision as for the unoptimized binaries. However, Table 8 shows that the least correctly inferred compiler optimization setting is O2 for BinAlign. This is because of the optimization strategy followed by compiler that maintains a single copy for the common data and functions in the binary (i.e., COMDATs [29]). The optimization confines the alignment paddings, thereby affecting the inference of the highly optimized binaries.

However, the improved BinAlign infers the class label for O1 and Ox optimized binaries with a reasonably better score due to the reason that the optimizing compilers on Windows maintain separate copies of the multiple definitions of the aligned functions and data at Ox. The code generated by O1 optimization level remains consistent throughout all the compilers. This is because the compilers maintain most of the alignment padding in the data sections to favor 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 of real-world binaries.

Table 8 – The result of four-levels of compiler optimization prediction.

	o-glassesX			BinAlign			Improved BinAlign		
Class label	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 9 – The result of Joint source compiler with two-levels of compiler optimization prediction.

	o-glassesX			BinAlign			Improved BinAlign		
Class label	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

6.3 RQ3. How effective is BinAlign in identifying the joint source compiler and optimization level of a Windows binary?

Tables 9 and 10 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 9, 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.

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. One of the challenges for deep learning model is the highly optimized code and the inter-procedural optimization in optimized binaries. Moreover, we can see that the performance of all neural network models for the unoptimized and the most-speedy optimization is comparatively better for MSVC and ICC. This is because the optimizing compilers support advanced vector extension (AVX) architecture that generates over-aligned instructions for increased performance [17]. Also, our test set instances comprise of floating point instructions, 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

Table 10 – The result of Joint source compiler prediction with four-levels of compiler optimization.

	o-glassesX			BinAlign			Improved BinAlign		
Class label	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	0.888	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	0.807
Clangcl-Od	0.981	0.9808	0.9809	0.9883	0.985	0.9866	0.9932	0.9816	0.9874
Clangcl-O1	0.6215	0.4529	0.524	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	0.995	0.9943	0.9947	0.9941	0.9979	0.996
ICC-O1	0.7487	0.5993	0.6657	0.7614	0.7666	0.764	0.8759	0.8062	0.8396
ICC-O2	0.3853	0.1187	0.1815	0.6586	0.089	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	0.964	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

characteristics of the deep learning CNN models is its higher performance and better accuracy for the zero-padded data bytes [22]. 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.

6.4 Malware Case Study

To illustrate the generality of our approach for compiler prediction of real-world malwares, we compile 113 C/C++ source code of Win32 malware downloaded from theZoo[7] 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. 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 are the same as mentioned in Section 5, however for the current evaluation we compile them with MSVC compiler and three different versions of the MSVC compiler.

Tables 11 and 12 show the malware attribution of the source programs and the distribution of our malicious dataset, respectively. We gather seven different families of malware programs in C/C++ language as shown in Table 11. The malware programs are then compiled with three versions of MSVC compiler in four different optimization settings as shown in Table 12, with the most successful compilation in version 2019 that supports the most APIs. Table 13 shows the overall classification result of compiler version prediction and compiler optimization prediction of our malware dataset. Our results demonstrate that the

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

Table 11 – 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 12 – The number of malware executables and dlls 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

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.

6.5 Custom Alignment Padding

Considering another scenario when software developers or malware authors intentionally modify the compiler's default alignment settings, we conduct a case study comprising of the binaries that enforce custom alignment paddings 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. We found that MSVC, Clangcl and ICC support four major alignment settings, defined as, i) `ALIGN`, ii) `FILEALIGN`, iii) `Zc:alignedNew`, and iv) `Zp16`, corresponding 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 [29]. 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 [4]. Thus, we train the models with custom alignment of 8192 bytes for all the alignment options discussed above.

Table 13 – The overall result of malware compiler version prediction for MSVC compiler, and compiler optimization prediction at four different levels.

	Compiler Version			Compiler Optimization		
Metrics	o-glassesX	BinAlign	Improved BinAlign	o-glassesX	BinAlign	Improved BinAlign
Prec.	0.8540	0.8713	0.9160	0.8718	0.8825	0.9287
Rec.	0.8548	0.8729	0.9170	0.8740	0.8827	0.9295
F	0.8542	0.8719	0.9162	0.8722	0.8822	0.9290

Table 14 – The overall prediction result of source compiler and compiler optimization level of custom-aligned binaries.

	Source Compiler			Compiler Optimization		
Metrics	o-glassesX	BinAlign	Improved BinAlign	o-glassesX	BinAlign	Improved BinAlign
Prec.	0.7923	0.8280	0.8557	0.6892	0.7708	0.7950
Rec.	0.7927	0.8294	0.8548	0.6729	0.7709	0.7920
F	0.7925	0.8287	0.8552	0.6811	0.7709	0.7935

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 paddings on the same dataset as evaluated in Sections 6.1 to 6.3. We then test our trained models with custom-aligned binary code to measure their evaluation performance. Therefore, we utilized 70% of our total compiled data for training, while 30% of the custom-aligned binaries are evaluated by the trained models.

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 14.

7 Limitations

Based on our evaluation of compiler provenance models, we state a limitation in our evaluation approach.

We evaluate BinAlign on selected options of custom alignment padding provided by the compiler. For example, the alignment options in MinGW compiler for aligning data for the C++17 standard, functions, loops, jumps and labels range from 2^2 to 2^{16}. Moreover, it would be interesting to assign labels to differently padded binary code with varying alignment padding compiler settings. BinAlign can thus be learnt on the padded binary data and evaluate it to predict the class of the alignment option set by the developers. We leave this as future work. Moreover for our current evaluation, we only considered the compiler specific alignment options. This work can further be extended to incorporate the architecture-specific alignment padding options [4].

8 Conclusion

In this work, we evaluate 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, our proposed model explores the alignment padding in the binary code and infers the source compiler and optimization levels over a diverse set of real-world and complex binaries. Our experimental results show that leveraging data and code alignment pattern in binary code effectively improves the performance of state-of-the-art compiler provenance model. Our evaluation results further show that BinAlign effectively improves the compiler provenance analysis of a large set of 64-bit benign and 32-bit malicious binaries.

We believe that with the growing strategies followed by optimizing compilers, new patterns of instructions may be generated to serve the purpose of alignment padding. Hence this work can be extended to incorporate additional filler instructions in order to enhance the performance of compiler provenance analysis in optimized binaries.

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. Intel compatability with msvc. https://support.alfasoft.com/hc/en-us/articles/360002874938 (2019), accessed: 2021-07-01
2. Clangcl with msvc. https://clang.llvm.org/docs/MSVCCompatibility.html (2021), accessed: 2021-07-01
3. Detect-it-easy. https://github.com/horsicq/Detect-It-Easy (2021), accessed: 2021-07-01
4. Mingw-w64 compiler. https://www.mingw-w64.org/ (2021), accessed: 2021-11-19
5. Portabl executable identifier. https://www.aldeid.com/wiki/PEiD (2021), accessed: 2021-07-01
6. Ransomware. https://www.statista.com/statistics/701020/major-operating-systems-targeted-by-ransomware/ (2022), accessed: 2022-03-09
7. Windows, the most popular desktop os. https://gs.statcounter.com/os-market-share/desktop/worldwide (2022), accessed: 2022-03-08
8. BinAlign Appendix (2023), https://anonymous.4open.science/r/CompProv_Feb2023
9. Andriesse, D.: Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly. No Starch Press, Incorporated (2018), https://books.google.com.sg/books?id=laWgswEACAAJ
10. 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)
11. 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)
12. 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)
13. Cifuentes, C., Gough, K.J.: Decompilation of binary programs. Software: Practice and Experience **25**(7), 811–829 (1995)

14. 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)
15. Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. No Starch Press, USA (2011)
16. Grune, D., Van Reeuwijk, K., Bal, H.E., Jacobs, C.J., Langendoen, K.: Modern compiler design. Springer Science & Business Media (2012)
17. Intel: Intel 64 Optimization Reference Manual. Intel Corporation (2016), https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf
18. Ji, Y., Cui, L., Huang, H.H.: BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network, p. 702–715. New York, NY, USA (2021)
19. Ji, Y., Cui, L., Huang, H.H.: Vestige: Identifying binary code provenance for vulnerability detection. In: Applied Cryptography and Network Security (2021)
20. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
21. Koo, H., Park, S., Choi, D., Kim, T.: Semantic-aware binary code representation with bert. arXiv preprint arXiv:2106.05478 (2021)
22. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Communications of the ACM **60**(6), 84–90 (2017)
23. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: International Workshop on Recent Advances in Intrusion Detection. pp. 207–226. Springer (2005)
24. 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)
25. Lin, Y., Gao, D., Lo, D.: Resil: Revivifying function signature inference using deep learning with domain-specific knowledge. In: Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy. pp. 107–118 (2022)
26. LLVM: LLVM Documentation. LLVM Compiler Infrastucture (2020)
27. 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)
28. Massarelli, L., Di Luna, G.A., Petroni, F., Baldoni, R., Querzoni, L.: Safe: Self-attentive function embeddings for binary similarity. In: Detection of Intrusions and Malware, and Vulnerability Assessment (2019)
29. Microsoft: Visual C++ Compiler Optimization Documentation (2021), https://docs.microsoft.com/en-us/cpp/build/reference/compiler-options-listed-alphabetically?view=msvc-160
30. Microsoft Corporation: Developer Documentation (2021), https://docs.microsoft.com/en-us/windows/win32/debug/pe-format
31. Mitchell, M., Oldham, J., Samuel, A.: Advanced linux programming. New riders Berkeley, CA (2001)
32. Muchnick, S., et al.: Advanced compiler design implementation. Morgan kaufmann (1997)
33. 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)
34. 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)

35. 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)
36. Rahimian, A., Nouh, L., Mouheb, D., Huang, H.: Binary code fingerprinting for cybersecurity
37. Rahimian, A., Shirani, P., Alrbaee, S., Wang, L., Debbabi, M.: Bincomp: A stratified approach to compiler provenance attribution. Digital Investigation (2015), the Proceedings of the Fifteenth Annual DFRWS Conference
38. Ramshaw, M.J.: Establishing malware attribution and binary provenance using multicompilation techniques (2017), https://www.osti.gov/biblio/1390004
39. 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 '11 (2011)
40. 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 '10 (2010)
41. Rosenblum, N.E., Zhu, X., Miller, B.P., Hunt, K.: Learning to analyze binary computer code. In: AAAI. pp. 798–804 (2008)
42. 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)
43. Tian, Z., Huang, Y., Xie, B., Chen, Y., Chen, L., Wu, D.: Fine-grained compiler identification with sequence-oriented neural modeling. IEEE Access (2021)
44. TIS Committee: Executable and Linking Format (ELF) Specification (1995), https://refspecs.linuxfoundation/elf/elf.pdf
45. Wang, R., Shoshitaishvili, Y., Bianchi, A., Machiry, A., Grosen, J., Grosen, P., Kruegel, C., Vigna, G.: Ramblr: Making reassembly great again. In: NDSS (2017)
46. Wang, S., Wang, P., Wu, D.: Reassembleable disassembling. In: 24th USENIX Security Symposium. pp. 627–642. Washington D.C. (2015)
47. 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)
48. Xue, H., Sun, S., Venkataramani, G., Lan, T.: Machine learning-based analysis of program binaries: A comprehensive study. IEEE Access **7**, 65889–65912 (2019)
49. Yang, S., Shi, Z., Zhang, G., Li, M., Ma, Y., Sun, L.: Understand code style: Efficient cnn-based compiler optimization recognition system. In: ICC 2019 - 2019 IEEE International Conference on Communications (ICC) (2019)
50. 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)

Appendix

o-glassesX Architecture. o-glassesX 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 [34]. 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 zero's to construct a 2048-bit sequence of 16 instructions. The underlying architecture of neural network comprises of the following CNN layers. (1) The first layer *instructionCNN* that takes 2048 bits as input. Each unit is a single-dimension 128-unit kernel, with the stride of 128 and having depth of 96. (2) The second layer is the positional encoding (PE) layer with 256 filters to a 16 x 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 in two dimensions by setting the stride and kernel size to 1. (3) 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 [34].

Different locations where compilers emit alignment padding. Table 15 [8] lists nine scenarios in which compilers emit alignment padding at different locations in the binary.

Github projects in our dataset. Table 16 [8] shows the description of some of the projects in our dataset.

Table 15 – Placement of alignment padding in the binaries. We identified nine scenarios in which compilers emits alignment padding at different locations in the binary

#	Placement in Binary	Purpose and Location	Impact of Optimization	Align
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 functions	aligned jump tables	less in MSVC & Clangcl, intense in ICC at O2, Ox [17]	NOP
4	Intrinsic functions	compiler's inlined functions	expansion of function vary with compiler family & optimization [26]	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

Table 16 – 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 library	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.