DIComP: Lightweight Data-Driven Inference of Binary Compiler Provenance with High Accuracy

Ligeng Chen†, Zhongling He†, Hao Wu†, Fengyuan Xu†, Yi Qian† and Bing Mao†
†State Key Laboratory for Novel Software Technology, Nanjing University
{chenlg, zhe, hako.wu, yi_qian}@smail.nju.edu.cn, {fengyuan.xu, maobing}@nju.edu.cn

***Abstract*—Binary analysis is pervasively utilized to assess software security and test vulnerabilities without accessing source codes. The analysis validity is heavily influenced by the inferring ability of information related to the code compilation. Among the compilation information, compiler type and optimization level, as the key factors determining how binaries look like, are still difficult to be inferred efficiently with existing tools. In this paper, we conduct a thorough empirical study on the binary's appearance under various compilation settings and propose a lightweight binary analysis tool based on the simplest machine learning method, called DIComP to infer the compiler and optimization level via most relevant features according to the observation. Our comprehensive evaluations demonstrate that DIComP can fully recognize the compiler provenance, and it is effective in inferring the optimization levels with up to 90% accuracy. Also, it is efficient to infer thousands of binaries at a millisecond level with our lightweight machine learning model (1MB).**

***Index Terms*—Binary Analysis, Compilation Options**

I. Introduction

The security community has been putting many efforts into designing binary analysis techniques, e.g., vulnerability retrieval [1], reverse analysis [2], [3], and vulnerability reproduction [4], [5]. Continuously improving the performance of the analysis techniques is a demanding requirement to protect system security. These techniques commonly take the semantic and structural information (e.g., control flow or data flow) from the binary as input, extract specific features, and perform the analysis. They assume that the features extracted from binary are mainly determined by the program's logic. However, we hold that the *binary's appearance* (i.e., the initial state of the binary code) is determined by both the program itself and the compilation options together. Neglecting the impact of the compiler may impede the effectiveness of the above tasks.

We investigate the following tasks and find that their results are considerably enhanced by taking into account the provenance of compiler and optimization level. ❶The basic block level binary comparison [6], [7] can not achieve a good result on binaries compiled at different optimization levels. Indeed, we find that twice as many basic blocks are generated at -O0 as -O3. So it would be better for us to identify the optimization level of the binaries first, and then we can retrieve the target binary [1] in the corpus. ❷And the same instruction under different compilation options may refer to different types of variables, which can affect the accuracy of type inference [8], [9]. ❸What's more, Mu [4] shows that vital information is always missing to reproduce the vulnerabilities in the real world. We can figure out the program configurations [10] with much higher confidence when holding the compilation options. The evidence above motivates our work, and the tasks above will be elaborated in Section II.

According to the investigation above, we find that the type of the compiler and the level of optimization have a significant impact on the binary analysis, which will also be reflected in the binary's appearance. To verify how they influence the appearance, we conduct a comprehensive analysis. Binary code is hard to understand directly. By utilizing the disassembler IDA Pro [11], we can transform the binaries into assembly code. With the help of the distribution of mnemonics and registers, we can figure out some differences (e.g., *-O1* of GCC uses more *callq* frequently than other optimization levels). Fortunately, some works [12], [13] leverage deep learning to identify the function boundary of binaries, which bring us a new feature — function length to measure the appearance. The result shows that the average function length of *-O3* is longer than *-O2* of GCC, which may be caused by some loop unrolling optimization options.

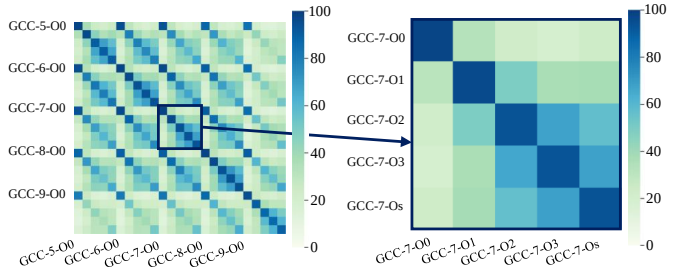
To scope our study, we comprehensively study the appearance of different compilers (GCC, Clang), optimization levels (-O0, -O1, -O2, -O3, -Os), and compiler versions (5 main versions for GCC, 7 for Clang). According to the observation from the study, we develop a lightweight tool called DIComP (***D*ata-driven *I*nference of binary *Com*piler *P*rovenance**). By leveraging the distribution of mnemonics, registers, and function length, we can distinguish the compilers and infer the optimization levels with high accuracy. Unfortunately, due to the high similarity between different compiler versions, we just classify the part of GCC versions and can do less with the versions of Clang. The main contributions are as follows:

1. We comprehensively study how the different compilation options influence the appearance of binaries in terms of mnemonic, register, and function length. (Section III)
2. We design and implement a lightweight and efficient system called DIComP, which can infer the provenance of compiler and optimization level. (Section IV)
3. We accomplish a comprehensive evaluation of a wide range of applications, and the result shows great performance on accuracy and processing speed. (Section VI)

II. Motivating Examples

In this part, we show three motivation examples to illustrate how compilation options affect the binaries' appearance.

Case 1: Binary Similarity. Binary similarity analysis [14], [15] aims to measure the similarity between pairs of binaries [1], [6], [7], which is often used in malware detection [1]. A precise similarity measurement can bring better detection results. Here, we experiment with a widely-used similarity-measurement tool BinDiff [7]. As shown in the Figure 1, we cross-compare the binaries compiled from *libxml* with 25 different combinations of compilation options (5 versions of GCC $\times$ 5 optimization levels). Figure 1a) presents the overall result, and Figure 1b) elaborates the result in the same compiler version. The darker the grids, the more similar they are and vice versa. It shows that the optimization level has a great impact on the binary similarity comparison, while the version has less influence.



a) Similarity of Same Compiler (GCC) b) Similarity of Same Version (GCC 7.0)

Fig. 1: Similarity of binaries compiled from the same source code with different optimization levels.

Case 2: Type Inference. Type inference [8], [9] aims to reconstruct the variables' types from the binary code, which is a requirement of reverse engineering [3]. Existing type inference efforts do not take into account the impact of the compilation, and we find that ignoring the compiler's impact can affect the inference result. For example, the instruction *'lea 0xOFFSET(%rsp), %rax'* can respectively represents the operation of variable type *point-to-char* and *struct* in different instruction contexts when the binary is compiled with different compilation options. If we know the compilation options, we can achieve better inference results. We evaluate how the type inference [8] benefit from DIComP, result of which is shown in Table I. The results show that knowing the compilation options can improve the inference accuracy (ACC) by 6.5%.

TABLE I: Fine-grained result of inferring variable types. Column 1 and 2 are the compilation options.Column 3 is the number of variables to infer. Column 4 is the inference results without knowing the compilation results. Column 5 is the inference results when the compiler type is given. Column 6 is the inference results when both the compiler type and optimization level are given.

		SUPPORT	ACC (Mixed)	ACC (C.)	ACC (C.& O.L.)
GCC	O0	102394	0.721	0.779	**0.802**
	O1	20238	0.613	0.624	**0.642**
	O2	19007	0.567	0.568	**0.579**
	O3	18135	0.566	0.557	**0.566**
	Total	159774	0.669	0.709	**0.728**
Clang	O0	105660	0.722	0.739	**0.747**
	O1	3159	0.525	0.525	**0.525**
	O2	2893	0.457	0.464	**0.473**
	O3	2883	0.462	0.463	**0.468**
	Total	114595	0.703	0.719	**0.727**

Case 3: Vulnerability Reproduction. Vulnerability reproduction is the first step to diagnose program failure. However, current works [4], [5] show that reproduction is challenging because of the lack of vital building configurations to perform the compilation. Some vulnerabilities can only be reproduced by enabling some specific configuration during compilation. For example, CVE-2018-9251 [16] can only be triggered by turning on '-with-lzma' under specific compilation options. Ensuring the consistency of compilation options as much as possible can greatly help users to reproduce the vulnerability. We experiment to identify the problematic configurations on 21 configuration-related vulnerabilities [17] from 4 well-known applications (i.e., libxml, OpenSSL, PHP, proftpd). Reproduction of this kind of vulnerability contains 3 steps: compilation option inference, configuration inference, locating vulnerability. Our tool dramatically compresses the search space of compilation options' combination, which saves time for compilation option inference. On average, it helps saving half of the processing time.

III. Exploratory Analysis

We focus on analyzing stripped binaries. Due to the low readability of stripped binaries and the limited information provided, we focus on how the compilation options affect the distribution of mnemonics, registers, and function lengths.

Data set. We create a comprehensive training data set from several open-source software projects, over 28,000 binaries in total. Here we enumerate some projects of different categories: OS tools (coreutils, binutils, etc), network programs (PHP, nginx, etc), computationally intensive programs (pdf, zlib, etc), and projects like Python which integrate packages of different categories. In total, over 28,000 binaries are used for analysis. We choose popular projects that reveal the distribution of appearance in reality. For the need of investigation, we build each project with different optimization levels (*-O0* to *-O3* and *-Os*), and with different versions of GCC (version 5, 6, 7, 8, 9) and Clang (version 3.9, 4, 5, 6, 7, 8, 9).

We choose GCC [18] and Clang [19] because they are the most widely used compilers at present. And we also choose different versions that are currently in use.

In the following subsections, we will control the variables and discuss how the different compilation options influence the distribution of mnemonics, registers, and function lengths.

A. Compiler

Firstly, we investigate how the compiler influences the appearance of binaries. We take GCC and Clang as the targets because they are the most popular compilers among open source projects. GCC is a part of the GNU toolchain, which has a large number of users. And almost all the Linux software supports the compilation using GCC. As a rising star, Clang has a more complete system design and a large number of community contributors. And Clang can generate intermediate code from the source code. So that many security analysts use Clang as the front end for program analysis.

To focus on the compiler, we leave all the other options aside. We divide our data set into two parts — compiled from GCC and compiled from Clang. That is to say, the part compiled from GCC is mixed by the binaries compiled from

Fig. 2: Distribution of mnemonic (left) and registers (right) divided by the granularity of compiler, optimization level, and compiler version.

different optimization levels and compiler versions of GCC So as Clang.

Mnemonics. As we can see on the left side of Figure 2, GCC and Clang seem to have a similar usage of mnemonics. So we list the mnemonics in Table II, of which frequency of appearance is in the top 5 (left side), and which have the most different distribution (right side) between GCC and Clang.

For example, 36.954% for mov indicates that mnemonic mov represents 36.954% of all the mnemonics which are collected from the binaries in the dataset compiled from GCC. Columns 1 to 3 show the most frequently appearing mnemonics. As for the mnemonic lea, it appears much more frequently in GCC than Clang. Column 4 to 6 present the mnemonics which have the most different distribution of GCC and Clang. We take movabs and movups as the example, and both of them hardly appear in the binaries compiled by Clang. But they only appear one fth the frequency in the binaries compiled by GCC, which may be an obvious feature to identify the compiler. Although the distribution of mnemonics between GCC and Clang is slightly different in the gure, we still can mine the in uence of compiler from the data listed in the table.

TABLE II: Selected mnemonics partially in uenced by compiler.

Mnemonic	GCC	Clang	Mnemonic	GCC	Clang
mov	36.954%	37.060%	xor	2.190%	2.823%
callq	6.70%	6.574%	jmpq	2.817%	3.322%
lea	5.333%	3.054%	add	3.293%	3.775%
je	4.210%	3.856%	movabs	0.101%	0.510%
cmp	3.741%	4.079%	movups	0.073%	0.428%

Registers. Mnemonic plays the role of an operator, while the register acts as a container. As shown on the right side of Figure 2, we can roughly observe the statistical results of register distribution. The gure presents the obvious differ ence between compilers, such as that GCC uses %eax more frequently than Clang. We classify the registers by their usage and discuss how the compiler in uences them respectively.

As shown in Table III, some differences are easier to be found in the table rather than in the gure. Due to the limited space of the paper, we only present some registers which can obviously show the difference between compilers. Row 2 to 4 denote the registers used to pass parameters, row 5 and 6 denote the pointer registers used to maintain the original data, row 7 denotes the miscellaneous registers saved by the caller, row 8 and 9 denote the miscellaneous registers saved by the callee, row 10 to 13 denote the general registers. Some registers can be partially used, which appear as a new name. Take the counting register %rcx as an example, which is a 64-bit register. Register %ecx is the 32 bits version with similar functionality, of which %cx is the lower 16 bits %ch and %cl is the higher 8 bits and the lower 8 bits of %rcx. Meanwhile, registers %r8, %r8d, %r8b, %r8w have the same relationships with the registers above. We group these registers into their own family, and we select some of them which can show the diverse distribution of different compilation options.

TABLE III: Selected registers partially in uenced by compiler.

Register	GCC	Clang	Register	GCC	Clang
%ecx	1.835%	3.640%	%edx	3.680%	2.905%
%rcx	2.854%	5.640%	%rdx	5.871%	3.731%
%cl	0.298%	0.737%	%cx	0.039%	0.103%
%ebp	0.597%	1.086%	%rbp	9.923%	11.054%
%rsp	5.629%	5.139%	%rbx	5.106%	5.327%
%r10d	0.238%	0.153%	%r11d	0.178%	0.113%
%r12	2.164%	2.058%	%r14	1.487%	2.936%
%r13	1.667%	1.782%	%r15	1.403%	2.469%
%cs	0.219%	0.500%	%eax	9.065%	6.960%
%ds	0.049%	0.002%	%rax	19.868%	15.666%
%es	0.087%	0.003%	%rip	3.603%	2.246%
%fs	0.318%	0.010%			

GCC and Clang have different preferences in choosing registers. According to the table, Clang prefers using %ecx, %rcs, %cl to pass parameters, while GCC prefers using %edx, %rdx, %cx. As for account of %eax and %rax, GCC uses them more frequently than Clang. The segment register is rarely used for both of GCC and Clang. But we have a much lower probability of seeing %ds and %es in Clang compiled binaries.

Expansion Rate. COTS (commercial off-the-shelf) binaries are typically stripped of much information about the source code. According to the different compiler behaviors, optimization options, and the version updated, the size of the binaries appears to be different. That is to say, a line of source code may be compiled into different lines of assembly code.

To study quantitatively, we raise a new concept called assembly expansion rate, which is de ned as follows,

$$(c_s, o_t) = \sum_{i=1}^{n} \frac{LoA(p_i, c_s, o_t)}{LoS(p_i)} \quad (1)$$

As shown in the formula, (c_s, o_t) denotes the assembly expansion rate of the concrete compiler c_s and optimization level o_t. $LoA(p_i, c_s, o_t)$ denotes the lines of the assembly code which is compiled from the program p_i with compiler c_s and optimization level o_t. $LoS(p_i)$ denotes the lines of program p_i's source code. In this work, we use a dataset $P = \{p_1, p_2, ..., p_n\}$, a compiler set $C = \{GCC, Clang\}$, and a optimization level set $O = \{O0, O1, O2, O3, Os\}$. In this part, we leave the compiler version aside which has little influence on the statistical result. So we compile the source code with different versions of compiler, and average the result.

The expansion rate of stands for the mapping relationships between lines of source code and lines of assembly code with different compilation options. The smaller the rate, the more the compiled binary is compressed.

TABLE IV: Mapping relationship between assembly code and source code on different compilers and optimization levels.

Compiler	O.L.		Compiler	O.L.	
GCC	O0	3.6747	Clang	O0	3.8167
GCC	O1	2.1508	Clang	O1	2.1324
GCC	O2	2.2491	Clang	O2	2.4421
GCC	O3	2.7876	Clang	O3	2.6085
GCC	Os	1.9298	Clang	Os	2.0665

Table IV shows the expansion rate of different compilers and optimization levels. The result of GCC is presented on the left, and the result of Clang is on the right. GCC has a similar expansion rate of with Clang in the same level of optimization. The expansion rate of both of them is also very similar in the changing trend of the optimization level. The expansion rate of -O0 is the largest under both compilers because it has the least optimization options. The sudden decrease in the expansion rate of -O1 is also due to the increase of optimization options. The expansion rate of -O2 and -O3 is relatively increased because the compilation takes the optimization of the run-time into account and may expand the loop. Optimization level -Os is special because it considers the size of the binary, so it has the smallest expansion rate under both GCC and Clang.

B. Optimization Level

We now investigate how binaries behave under different optimization levels. According to the presented result in the previous section, GCC and Clang have different system designs, which leads to the different distribution of mnemonics and registers. In this section, we concentrate on the impact caused by optimization levels. We study the distribution of mnemonics, registers, and function length compiled from different optimization levels under the same compiler.

a) Optimization Levels of GCC: According to the official documents, GCC has many optimization levels with different purposes, and there are associations between them. Besides the default option -O0, -O1, -O2 and -O3 consists of different sets of optimization options. There are also -Ofast which disregards strict standards compliance and -Og which optimizes the debugging experience.

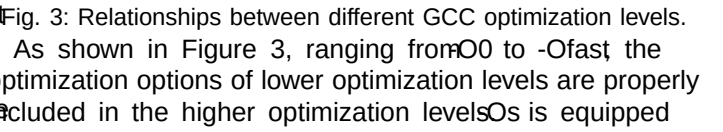


Fig. 3: Relationships between different GCC optimization levels.

As shown in Figure 3, ranging from -O0 to -Ofast, the optimization options of lower optimization levels are properly included in the higher optimization levels. -Os is equipped with all the options of -O2 but also some more options aiming to tune for code size rather than execution speed. According to the usage frequency, we select 5 mainstream optimization levels for analysis, which are -O0, -O1, -O2, -O3 and -Os.

Mnemonic. According to official GCC documentation, we find that -O1 has 45 more options than -O0, -O2 has 48 more options than -O1, -O3 has 16 more options than -O2, and -Os has 6 more options than -O2. It seems that, among the neighboring optimization levels, -O2 and -O1 are the most different, and -O2 and -Os are the hardest to be classified, due to that the number of different options directly influence the difference on the appearance. However, after we conduct a coarse-grained survey, we find that -O2 and -O3 are most similar. The reason for the phenomenon is that -O3 adopts many vectorizing algorithms to improve the parallel execution of the code, but its optimization conditions are relatively harsh, leading to the frequency of occurrence is relatively low. Indeed, for some programs, the result of compilation under -O2 and -O3 is identical.

TABLE V: Percentage of selected mnemonics at different GCC optimization levels.

Mnemonic	G-O0	G-O1	G-O2	G-O3	G-Os
mov	50.084%	35.069%	31.514%	31.053%	31.160%
callq	5.721%	7.947%	7.009%	6.520%	6.991%
lea	4.188%	6.075%	5.527%	5.713%	5.739%
je	2.855%	4.686%	4.655%	4.829%	4.650%
cmp	2.188%	4.133%	4.098%	4.522%	4.446%
pop	0.575%	2.507%	3.281%	2.835%	3.060%
push	1.297%	2.667%	2.520%	2.192%	2.501%
xor	0.648%	0.870%	3.463%	3.151%	3.353%
cmp	2.188%	4.133%	4.098%	4.522%	4.446%
movdqa	0.008%	0.015%	0.026%	0.252%	0.229%

To better observe the difference between the optimization levels, we mainly selected some high-frequency mnemonics and the mnemonics with relatively large discrimination between -O2 and -O3. As shown in Table V, we select 10 mnemonics from the data set. Column 2 to 6 show the distribution of mnemonics of -O0, -O1, -O2, -O3 and -Os from GCC. We can find that -O0 appears very different from the rest of the optimization levels, mainly because it is the default optimization level to reduce compilation time and make debugging produce the expected results. As -O1, it appears much similar to others, but we can still find some differences in the distribution of pop and xor. According to the table, we can hardly find the difference between -O2, -O3, and -Os. There is only slight difference between -O2 and -O3 on callq,

cmp and movdqa. Due to the optimization purpose of -Os, it prefers to use mnemonics for code space compression.

Register. As shown in Table VI, column 2 to 6 separately shows the distribution of registers of -O0, -O1, -O2, -O3 and -Os. Optimization level -O0 still stands out from others. It prefers to leverage %rbp for passing pointer rather than %rsp. Higher optimization levels optimize the size of the binary code, so they utilize much more %rsp to maintain the structure of the stack. For the general register, -O0 employs %rax at about one-third of registers to pass arithmetic, while other optimization levels prefer to use more diverse registers and optimize the process of arithmetic passing. Although -O1 has no obvious distinction, we can still see some differences from the comparison with -O0 and -O2. On the other hand, -O2 and -O3, there is little difference in the distribution of registers. Only in Row 8 to 10 can we figure out some little difference between them. It is because the optimization options of -O3 optimizes the inline functions. Optimization level -Os almost appears the same as -O2 in the register table, while it still has some difference with -O2 on the distribution of mnemonics.

TABLE VI: Selected registers of different GCC optimization levels.

Register	G-O0	G-O1	G-O2	G-O3	G-Os
%ecx	1.084%	1.878%	2.116%	2.260%	2.256%
%rcx	2.148%	3.193%	3.068%	3.133%	3.179%
%edi	0.682%	1.675%	1.738%	1.776%	1.830%
%rdi	3.904%	7.902%	7.175%	6.826%	7.273%
%edx	3.844%	3.472%	3.562%	3.688%	3.695%
%rdx	7.394%	5.597%	5.158%	5.199%	5.092%
%rbp	23.279%	4.988%	4.464%	4.211%	4.326%
%rbx	0.919%	7.674%	6.692%	6.375%	6.720%
%rsp	1.865%	7.881%	7.198%	6.934%	6.797%
%r10d	0.011%	0.163%	0.344%	0.410%	0.368%
%r11d	0.004%	0.129%	0.247%	0.305%	0.287%
%r12	0.204%	3.177%	2.976%	2.877%	2.874%
%r13	0.095%	2.334%	2.370%	2.309%	2.233%
%r14	0.099%	1.957%	2.124%	2.126%	1.983%
%r15	0.115%	1.824%	1.985%	2.005%	1.876%
%cs	0.020%	0.037%	0.481%	0.387%	0.251%
%eax	12.649%	7.856%	7.542%	7.400%	7.677%
%rax	30.676%	14.411%	16.692%	16.049%	14.472%

Function Length. As mentioned in the previous subsection, the expansion rate of of different optimization levels appears quite different. That is to say, one single line of source code maps to a different amount of binary code when compiled with different optimization levels. To align the code and maintain an appearance that can be easily understood by humans, we count the function length at the assemble code level, which can be easily generated from the binary code with the help of objdump

Fig. 4: Amount distribution of function length of binaries compiled from GCC with different optimization levels.

As shown in Figure 4, we can see the distribution of different lengths of functions (assembly-level) from binaries compiled with different optimization levels. Due to the limitation of the figure length, we merge some data. We take different intervals in points of 10, 40, 60, 140, 200, and 400, resulting in the abnormal peaks at 10, 40, and 60. There is also a peak at the length of 3, which is caused by dynamically linking library functions.

```
0000000000000640 <.plt>:
 640: ff 35 52 09 20 00        pushq   0x200952(%rip)
 646: ff 25 54 09 20 00        jmpq    *0x200954(%rip)
 64c: 0f 1f 40 00              nopl    0x0(%rax)

0000000000000650 <putchar@plt>:
 650: ff 25 52 09 20 00        jmpq    *0x200952(%rip)
 656: 68 00 00 00 00           pushq   $0x0
 65b: e9 e0 ff ff ff           jmpq    640 <.plt>

0000000000000660 <__stack_chk_fail@plt>:
 660: ff 25 4a 09 20 00        jmpq    *0x20094a(%rip)
 666: 68 01 00 00 00           pushq   $0x1
 66b: e9 d0 ff ff ff           jmpq    640 <.plt>

0000000000000670 <printf@plt>:
 670: ff 25 42 09 20 00        jmpq    *0x200942(%rip)
 676: 68 02 00 00 00           pushq   $0x2
 67b: e9 c0 ff ff ff           jmpq    640 <.plt>
```

Take the code above as an example. When we call a function from the library, we will first jump to table.plt (Procedure Linkage Table). Due to the mechanism of dynamic linking, when we first call the function printf, we jump to table.plt and fill the real address in the table.GOT (Global Offset Table) for use the next time. Thus, every time we call a library function, we will get a function with 3 lines of assembly code.

According to Figure 4, -O0 still behaves quite differently from other optimization levels. Functions with length of 1, 2, 4 can distinguish -O2 and -O3. But it can not provide much information for its limited amount. Besides, other parts of the figure, can not provide more features to distinguish between -O2 and -O3, which even confuse the boundary between -Os and them.

b) Optimization Levels of Clang: After thoroughly investigating the appearance influenced by different GCC optimization levels, we follow the same procedure to study the optimization levels of Clang. The relationships of different optimization levels of Clang are similar to the relationships of GCC. There is, however, a small difference that appears on -Os of Clang, which drops some optimization options from -O2. To compare with GCC, we select optimization levels -O0, -O1, -O2, -O3 and -Os from Clang for analysis.

We count the mnemonic. For the simultaneous comparison between compilers and optimization levels, we select mnemonics as described previously.

TABLE VII: Selected mnemonics of different Clang optimization levels.

Mnemonic	C-O0	C-O1	C-O2	C-O3	C-Os
mov	47.960%	33.357%	32.800%	32.574%	33.494%
callq	5.591%	8.123%	6.581%	6.458%	6.813%
lea	1.441%	3.440%	3.728%	3.734%	3.660%
je	2.466%	4.073%	4.481%	4.505%	4.360%
cmp	3.384%	3.564%	4.451%	4.626%	4.566%
jne	2.141%	2.832%	3.106%	3.317%	3.141%
test	0.357%	3.912%	4.301%	4.424%	4.196%
add	4.340%	3.409%	3.634%	3.707%	3.475%
jmpq	6.217%	2.225%	2.253%	2.284%	2.235%
jmp	0.030%	1.760%	1.774%	1.569%	1.728%

As shown in Table VII, column 2 to 6 show the distribution of mnemonics of -O0, -O1, -O2, -O3 and -Os from Clang.

-O0 behaves differently among all of the selected mnemonics. On the one hand, -O0 has a large amount of mov, which even takes up half. On the other hand, -O0 is only equipped with less amount of test and jmp than other optimization levels. As for -O1 and -Os of Clang, they still have slightly differences between others. Tracing back to Table VI, -O2 and -O3 of GCC have distinctions on the distribution of callq and cmp, while -O2 and -O3 of Clang seems have an even percent of these mnemonics, which really confuses both of human experts and machines. To figure out the root cause, we look up the document and find that -O2 and -O3 of Clang only have 2 different options. The reason above may explain why the appearance of -O2 and -O3 of Clang is so similar.

TABLE VIII: Selected registers of different Clang O.L.

Register	C-O0	C-O1	C-O2	C-O3	C-Os
%ecx	5.051%	2.743%	3.134%	3.165%	2.256%
%rcx	6.975%	4.419%	5.356%	5.361%	3.179%
%edi	1.028%	2.018%	2.072%	2.060%	1.830%
%rdi	5.621%	7.473%	6.528%	6.469%	7.273%
%edx	3.139%	2.590%	2.846%	2.829%	3.695%
%rdx	4.056%	3.221%	3.717%	3.727%	5.092%
%rbp	29.858%	3.624%	3.804%	3.864%	4.326%
%rbx	0.249%	8.332%	6.985%	6.939%	6.720%
%rsp	2.407%	6.603%	6.119%	6.072%	6.797%
%r10d	0.104%	0.125%	0.182%	0.184%	0.368%
%r11d	0.061%	0.087%	0.143%	0.145%	0.287%
%r12	0.068%	2.941%	2.807%	2.816%	2.874%
%r13	0.056%	2.317%	2.491%	2.515%	2.233%
%r14	0.067%	4.637%	3.873%	3.830%	1.983%
%r15	0.050%	3.737%	3.335%	3.294%	1.876%
%cs	0.271%	0.858%	0.591%	0.576%	0.251%
%eax	7.978%	6.748%	6.428%	6.448%	7.677%
%rax	19.741%	14.728%	14.286%	14.251%	14.472%

Register. According to Figure 2, ranging from -O1 to -Os of Clang has quite similar distribution of registers. To make the analysis complete enough for our study, we select exactly the same registers as Table VI to fill Table VIII. As shown in Table VIII, column 2 to 6 shows the distribution of registers of -O0, -O1, -O2, -O3 and -Os from Clang.

-O0 is again clearly distinguished from the others due to its default settings. As for -O1 and -Os, they also have some registers which have different distribution with each other, such as %ecx, %rdx, etc. So we try to dig more information on the aspect of the register, to classify -O2 and -O3. Unfortunately, as we can see in Table VIII, -O2, and -O3 of Clang nearly appear the same on the distribution of registers. We look through the whole data set to find some helpful information, but we fail.

Function Length. After failing to distinguish -O2 and -O3 of Clang with the help of mnemonics and registers, we turn for the last feature – function length. As shown in Figure 5, we can see the red and orange pillars stand out which represent the amount of function length of -O0 and -O1. As for mnemonics and registers, the function length does not help distinguish between -O2 and -O3, either.

C. Compiler Version

In the previous subsections, we investigate how the different compilers and optimization levels influence the appearance of binaries. In this part, we try to figure out how the appearance of binaries will change across different versions of the compiler. We select the mainstream versions of GCC (5, 6, 7, 8, 9) and Clang (3.9, 4, 5, 6, 7, 8, 9). The investigation result is not satisfied. As shown in Figure 2, no matter the distribution of mnemonics or registers, we can not summarize one single rule to distinguish them. We also analyze the distribution of function length, but still, nothing can help.

Fig. 5: Distribution of function length of binaries compiled from Clang with different optimization levels.

To further investigate the difference between different compiler versions, we leverage Bindiff to measure the similarity between the binaries compiled from different compiler versions. BinDiff is a binary comparison tool commonly used in industry. Since BinDiff itself does not have the capability to disassemble binaries, it is necessary to use IDA Pro to firstly disassemble the binary and generate an intermediate result to build graph structure for the binary. BinDiff is based on graph theory, looking for isomorphism graphs in two topological graphs to complete function similarity matching. Although the graph isomorphism problem is still an open problem in academia, BinDiff uses heuristic matching algorithms to make it acceptable in terms of both matching speed and accuracy.

TABLE IX: Similarity comparison between versions of GCC.

	G-5.0	G-6.0	G-7.0	G-8.0	G-9.0
G-5.0	99.2%	96.9%	94.1%	92.3%	82.5%
G-6.0	96.8%	99.1%	96.6%	95.0%	86.7%
G-7.0	93.8%	96.6%	99.1%	97.0%	90.0%
G-8.0	91.9%	95.0%	96.9%	99.2%	95.5%
G-9.0	82.0%	86.8%	89.9%	95.4%	99.2%

As shown in Table IX, we present results of cross-comparison between different versions, which is ranging from 0 to 100%. 96.8% in column 2 row 3 denotes that Bindiff think the binaries compiled by GCC version 5 and 6 are 96.8% similar. We analyze all the binaries in our data set and average them to fill the table.

Due to the symmetry of the table, the symmetric data should be completely consistent, and the data on the diagonal should theoretically be 100%. However, the result is not as expected. On the one hand, because the stripped binary does not contain debugging information, the disassembly tool cannot effectively rebuild all the information, resulting in loss of functions or information. On the other hand, it may be due to the strategy of Bindiff's matching algorithm, which caused the matching to fail or to match the wrong function.

TABLE X: Similarity comparison between versions of GCC.

	C3.9	C4.0	C5.0	C6.0	C7.0	C8.0	C9.0
C3.9	99.2%	97.7%	96.5%	95.2%	94.4%	94.0%	92.3%
C4.0	97.7%	99.2%	98.1%	97.0%	96.3%	95.8%	94.5%
C5.0	96.5%	98.1%	99.2%	98.2%	97.5%	97.1%	95.8%
C6.0	95.2%	97.0%	98.1%	99.2%	98.5%	98.1%	97.0%
C7.0	94.4%	96.3%	97.5%	98.5%	99.2%	98.7%	97.7%
C8.0	94.0%	95.9%	97.1%	98.1%	98.7%	99.2%	98.3%
C9.0	92.3%	94.5%	95.8%	97.0%	97.7%	98.3%	99.2%

According to Table IX, neighboring versions have a higher similarity, while versions with a large span share a lower

Fig. 6: An overview of the system work ow. The sub- gures separately represent the training phase and the testing phase.

similarity. Binaries of GCC version 5 are 96.8% similar to the GCC version 6, but they are only 82.0% similar to the GCC version 9. That is to say, we may distinguish the binaries compiled from version 9 from version 5, but the neighboring versions may mislead DIComP.

Also, we analyze the versions of Clang. As shown in Table X, the versions of Clang have a higher similarity between each other. Even the similarity between Clang version 3.9 with Clang version 9 is 92.3% (column 2 row 8). This makes it dif cult to distinguish the versions of Clang.

IV. Overview

In this section, we provide an overview of our method.

Given the analysis results above, we try to leverage machine learning to infer compilation options. According to Figure 6, our prototype consists of two phases, training and testing. The only difference between them is that we utilize debug information and the information of compilation options to train the Function Boundary Model and Hierarchical Model during the training phase, while we just input the stripped binaries and get the inference results during the testing phase.

Compile. To get a large number of binaries with different compilation options, we rstly compile projects using different compilers with different optimization levels and versions. We use a script to select a pre-built docker image [20] that contains different compilers. After compilation, we collect all the binaries contributing to the data set.

Extract. In the extraction stage, we use *objdump* to disassemble each binary, which can reach 99.4% of accuracy under our experiment. We split each assembly code into 3 elements: one mnemonic and two operands. Operands may be left blank if there are less than two operands. Then, we count both mnemonics and registers that appear in operands. To normalize our data, we divide the frequency of mnemonics and registers by the sum of their frequencies individually, so that we get the portion of each kind of mnemonic or register. We also count the length of the assembly code of each function and then get a normalized histogram of all the lengths. Finally, we put these features into a matrix for each binary for the next step.

Function Boundary Model. To get the length of each function, we need to locate the start point and the endpoint of the stripped binaries. Thanks to the previous works [12], [13], we can imitate the prototype of them to accomplish the mission. We use Word2Vec to transfer assembly language into vectors and mark those being the boundaries of functions. Its accuracy is very high which is almost 100%, and it can well assist the pipeline of inference.

Fig. 7: Structure of hierarchical model.

Hierarchical Model. Although the advanced machine learning method has enough tting capability to learn the pattern of each single compilation option with one model, we want to do it in a ne-grained way and make it more practical.

As shown in Figure 7, we present the structure of the hierarchical model. Firstly, we train a model to identify the compiler. Then, we separately train two models to distinguish the optimization levels of the binaries for GCC and Clang. At last, we have 10 sets of binaries, which are used to train their own models to classify the versions.

As mentioned in Section III, we have three sets of features that can be utilized, mnemonics, registers and function length, which can be represented as follows,

$$Feature(b_n) = \begin{cases} MNE(b_n) = N(mne_1; mne_2; :::; mne_i) \\ REG(b_n) = N(reg_1; reg_2; :::; reg_j) \\ FUNL(b_n) = N(funl_1; funl_2; :::; funl_k) \end{cases}$$

$MNE(b_n)$, $REG(b_n)$, and $FUNL(b_n)$ denote the matrices that are the statistical result of mnemonics, registers and function length for the binary b_n with the size of $1 \quad i$, $1 \quad j$ and $1 \quad k$. At different stages, we will use different combinations of features to infer the compilation options based on the experimental results.

Inference Unseen Binaries. Same as the training phase, we use *objdump* to disassemble unseen binaries. Then, we use the same techniques to extract portions of different mnemonics and registers. For function length, the function boundary model really helps.

At last, we feed the combination features from mnemonics, registers, and function length into our model and to infer the compilation options.

V. Implementation

In this section, we present the implementation of our system DIComP.

Model Selection. We aim to propose a practical tool to gure out the used compilation options from the stripped binaries. Security researchers can leverage the tool to improve the prior diagnosis for the security applications (e.g., vulnerability patch, reverse engineering). So it must have the following characteristics: ef cient , precise and lightweight.

Fig. 8: Control Flow over the different optimization levels compiled from the same source code.

Previously, several works [21]–[23] tried to solve this problem with traditional methods and learning-based methods, but none of them can meet all the needs. It is hard to balance the accuracy and ef ciency of the method.

Figure 8 above presents the control ow of the binaries compiled from the same source code but with different optimization levels. The blocks framed in red are referring to the same source code. It is obvious that the control ows indeed are slightly different locally and globally. But it can not be the most prominent feature to distinguish them.

So we take the distribution of mnemonics, registers, and function length as features, which also imply the relationship of control ow. The way of extracting and embedding the features is far more ef cient than before.

The learning-based method can achieve a great result. So we consider it to make it precise. But RNN-like (Recurrent Neural Networks) models [23], [24] and CNN (Convolutional Neural Networks) models [25] have a great number of parameters, which require numerous computational resources. According to our analysis result, MLP (Multi-Layer Perceptron) is powerful enough to capture the relationship between instruction frequency and the corresponding compilation options. Not only it can cross combine the features, but it is also very lightweight.

Processing Pipeline. We batch compile the source code for setting up the data set. DIComP disassembles the binaries with objdump. To gure out the distribution of mnemonics and registers, we develop a Python script that leverages regular expressions. According to the previous works [12], [13], we utilize LSTM (Long Short-Term Memory) Networks to learn the function boundary of the binary code which can help us know the distribution of function length. For the hierarchical inferring model, we use MLP as aforementioned to separately train each part. In all the processes equipped with the machine learning method, we use the Keras [26] package with the TensorFlow-CPU backend. For the evaluation part, we use the machine learning library scikit-learn [27] which calculates the metrics for each stage.

Hardware Equipment. All our experiments were conducted on a PC with 16GB memory, 1 Intel i7-4870HQ CPU (2.5 GHz), and 512GB SSD. We use the TensorFlow-CPU backend with Intel MKL support. To make our tool more friendly to use, all the experiments can be run on ordinary equipment.

VI. Evaluation

DIComP is a machine learning-based method, so we use three performance metrics commonly used to evaluate machine learning classi ers: precision (P), recall (R), and F1 score. Formally, they are de ned as follows:

$$P = \frac{TP}{TP + FP}; R = \frac{TP}{TP + FN}; F1 = \frac{2 \quad P \quad R}{P + R}$$

where TP is the true positives, FP is the false positives, FN is the false negatives. Precision is the ratio of cases where the predicted value is equal to the given value, which is the closeness of the measurements to each class. The recall is the proportion of correct predictions over the set of their class (i.e., the accuracy ratio of inferring the right compiler or optimization level). F1 score is a balance measurement that is calculated by precision and recall. All three metrics are in the range of 0 to 1.

To fairly evaluate our method, we split our data set into two parts, of which one part consists of 80% of binaries for training and another part consists of 20% of binaries for testing. There is no intersection between the two parts of the binaries, and they are from different applications.

A. Evaluation of distinguishing compilers.

TABLE XI: Classi cation result of compilers by mnemonic, register and function length.

	MNE			REG			FUNL		
	P	R	F1	P	R	F1	P	R	F1
GCC	1.00	1.00	1.00	1.00	1.00	1.00	0.85	0.26	0.39
Clang	1.00	1.00	1.00	1.00	1.00	1.00	0.70	0.97	0.81

According to the hierarchical model, we rstly distinguish the compiler of the binaries. As shown in Table XI, we present the classi cation results separately by using mnemonics (MNE), registers (REG), and function length (FUNL).

TABLE XII: Accuracy comparison with previous works on distinguishing the compilers.

	GCC	Clang
i2v_RNN [24]	0.99	0.97
Rosenblum2011 [21]	0.98	0.98
Rosenblum2010 [28]	0.93	-
DIComP	1.00	1.00

The experimental results con rm our previous observations. The binaries compiled from GCC and Clang have an obvious different distribution of mnemonics and registers, which can be utilized to distinguish the compiler, while they are similar to the distribution of function length. We individually utilize the mnemonics and registers, both of which can distinguish the compiler with 100% accuracy.

Table XII presents the comparison result with previous works. DIComP outperforms all the works. Rosenblum et al. [28] do not distinguish Clang at their work.

B. Evaluation of classifying optimization levels.

After we easily distinguish the compiler of the stripped binaries, we get two sets of binaries compiled from GCC and Clang. So we can classify them into corresponding optimization levels on the next stage.

TABLE XIII: Classification result of optimization levels (GCC) by mnemonic, register and function length.

		G-O0	G-O1	G-O2	G-O3	G-Os	Macro avg.
MNE	P	1.00	0.98	0.41	0.31	0.89	0.72
	R	0.99	0.97	0.53	0.19	0.98	0.73
	F1	0.99	0.98	0.46	0.24	0.93	0.72
REG	P	0.99	0.51	0.60	0.47	0.25	0.57
	R	1.00	0.89	0.36	0.64	0.07	0.59
	F1	1.00	0.65	0.45	0.54	0.11	0.55
FUNL	P	0.90	0.31	0.43	0.92	0.04	0.52
	R	0.96	0.95	0.07	0.63	0.00	0.52
	F1	0.93	0.47	0.12	0.75	0.00	0.45
MNE+REG	P	1.00	1.00	0.52	0.41	0.72	0.73
	R	1.00	0.92	0.47	0.34	0.98	0.74
	F1	1.00	0.96	0.49	0.37	0.83	0.73
MNE+FUNL	P	0.94	0.96	0.87	0.81	0.82	0.88
	R	0.99	0.95	0.75	0.74	0.96	0.88
	F1	0.97	0.96	0.81	0.77	0.89	0.88
REG+FUNL	P	0.99	0.47	0.76	0.84	0.38	0.69
	R	1.00	0.85	0.75	0.65	0.16	0.68
	F1	1.00	0.60	0.75	0.73	0.22	0.66
All	P	1.00	0.99	0.79	0.87	0.89	0.91
	R	1.00	0.96	0.85	0.74	0.97	0.90
	F1	1.00	0.97	0.82	0.80	0.93	0.90

Firstly, we evaluate the result of inferring the optimization levels from the binaries compiled by GCC. As shown in Table XIII, we present the result of using different combinations of features to infer the optimization levels. Column 2 to 6 separately show the result of classifying -O0, -O1, -O2, -O3 and -Os. Column 7 shows the average result of all the optimization levels. Merged row 1 to 3 shows the inferring results of using individual features, mnemonics (MNE), registers (REG), and function length (FUNL). Merged row 4 to 6 shows the inferring results of using combined features, and the last merged row shows the result predicted by all the features. According to the table, -O0, -O1, and -Os of GCC can be inferred from mnemonics' strength. We discover that binaries compiled with -O2 and -O3 of GCC have a quite similar appearance, which is hard for us to distinguish with individual features. Fortunately, when we combine the features of mnemonic and function length, we can roughly classify them. When we utilize all the features, the distinction between -O2 and -O3 will be greater, and the overall performance of inferring all optimization levels is also the best. The direct combination of the features does not produce the above result, all of which is due to the ability of the neural networks to mine the inherent connections of the features.

TABLE XIV: Accuracy comparison with previous works on distinguishing optimization levels of GCC.

	G-O0	G-O1	G-O2	G-O3	G-Os
HIMALIA [23]	0.99	0.99	0.73	0.74	0.99
BinEye [25]	0.98	0.97	0.98		0.96
BinComp [22]	0.91	-	0.91	-	-
Rosenblum2011 [21]	0.99		0.99		-
DIComP	1.00	0.96	0.85	0.74	0.97

Table XIV compares DIComP with previous works to distinguish the optimization level. Except HIMALIA [23], the rest of the previous works can not distinguish all the levels.

BinEye [25] groups -O2 and -O3 together. BinComp [22] only distinguishes -O0 and -O2. Rosenblum et al. [21] separately groups -O0 and -O1, -O2 and -O3. DIComP behaves a little worse at -O1 and -Os than HIMALIA, but the rest of the optimization levels outperforms all the tools.

TABLE XV: Classification result of optimization levels (Clang) by mnemonic, register and function length.

		C-O0	C-O1	C-O2	C-O3	C-Os	Macro avg.
MNE	P	1.00	0.89	0.00	0.50	0.64	0.61
	R	1.00	0.95	0.00	0.73	0.96	0.73
	F1	1.00	0.92	0.00	0.60	0.77	0.66
REG	P	0.98	0.59	0.35	0.32	0.77	0.60
	R	1.00	0.91	0.39	0.16	0.62	0.62
	F1	0.99	0.71	0.37	0.22	0.68	0.60
FUNL	P	0.81	0.33	0.43	0.49	0.80	0.57
	R	0.96	0.93	0.08	0.14	0.37	0.50
	F1	0.88	0.48	0.14	0.22	0.50	0.44
MNE+REG	P	1.00	0.86	0.51	0.53	0.86	0.75
	R	1.00	0.96	0.62	0.30	0.95	0.77
	F1	1.00	0.91	0.56	0.39	0.90	0.75
MNE+FUNL	P	1.00	0.50	0.53	0.51	0.80	0.67
	R	1.00	0.97	0.19	0.25	0.97	0.68
	F1	1.00	0.66	0.28	0.34	0.88	0.63
REG+FUNL	P	1.00	0.43	0.45	0.49	0.92	0.66
	R	1.00	0.95	0.24	0.14	0.90	0.65
	F1	1.00	0.59	0.32	0.21	0.91	0.61
All	P	1.00	0.57	0.46	0.46	0.95	0.69
	R	1.00	0.98	0.36	0.22	0.96	0.70
	F1	1.00	0.72	0.40	0.30	0.96	0.67

To verify the transferability of our method, we also accomplish the experiments to infer the optimization levels of Clang with the same experimental settings, the results of which are shown in Table XV. We can figure out that -O0 and -O1 can be easily distinguished from other optimization levels with the help of mnemonics. Different from GCC, the appearance of the binaries compiled from -O2, -O3, and -Os are more similar, which is hard to deal with. We can classify the binaries of -Os by leveraging all the features, but this is not sufficient to distinguish between -O2 and -O3. In particular, we find that the feature of function length is not able to distinguish -O2 and -O3 of Clang like GCC but even confuses the classifier. This result confirms that the similarity between the binaries compiled from -O2 and -O3 is very high because there have only two different optimization options.

Fig. 9: Confusion matrix of classifying optimization levels.

To further analyze the results, we introduce a confusion matrix to help us analyze why some binaries cannot be correctly classified. As shown in Figure 9, we present the confusion matrix of optimization levels of GCC and Clang. Take Figure 9 (a) as an example. The horizontal axis represents the ground truth of the binaries, and the vertical axis represents the value predicted by the model of the binaries.

TABLE XVI: Efficiency comparison with related works.

Method	Tech.	Training Time	Epoch	Testing Time	Params	FLOPS	Model Size
BinEye [25]	CNN	44Min50S	10	56S	1.31M	11.5M	16MB
HIMALIA [23]	Bi-GRU	143H20Min10S	10	3H7Min57S	2.37M	6.29M	23MB
DIComP (ours)	MLP	2.4S	10	0.06S	132K	1.31M	1MB

So the grids on the diagonal represent the binaries classified correctly. The darker the color, the greater the number of corresponding grids. Column 1 row 1 represents the amount of the binaries compiled from *-O0* of GCC which are correctly predicted to *-O0*, while column 1 row 2 represents that almost no sample of *-O0* is classified to *-O1*. According to the sub-figure (a), we can obviously figure out that most of the binaries compiled from *-O2* and *-O3* of GCC can be correctly classified, and the rest of them are not placed properly due to the similar optimization options between *-O2* and *-O3*. Pay attention to sub-figure (b), we originally regard the reasons for confusing the *-O2* and *-O3* of Clang the same as GCC. But the matrix tells that the binaries compiled from *-O2* of Clang can be classified to *-O3* and *-Os*, even *-O1*, which further illustrates that the similarity between optimization levels of Clang is higher than for GCC.

C. Discussion of compiler versions.

We try to infer the compiler version from the stripped binaries. But it seems that the results are not useful. Versions *3.9*, *6*, and *9* of Clang and versions *5* and *9* of GCC can be distinguished with accuracy of 52%, 59%, 74%, 95% and 66%. According to the result, we find that all of the versions of Clang seems more similar to each other, while GCC version 5 and version 9 has some obvious difference.

D. Efficiency.

Even though the data-driven method can *learn* the rules from a set of high-quality data, comparing with rule-based methods which saves much human effort, and it can iterate the system itself by adding the newly coming data. But it still faces the problem of the significant overhead of computational resources. We evaluate the overhead of DIComP compared with previous learning-based methods, to reflect our method is **efficient** and **lightweight** at the same time.

To evaluate fairness enough, we test all the methods on the same dataset, which contains 12,000 binaries from GNU, 80% for training, 20% for testing. To accelerate the procedure, we use 1 NVIDIA 1070 GPU in the experiment. The detailed evaluation result is shown in Table XVI. Column 2 denotes the learning model, column 3 and 5 separately denotes the training time and testing time. Column 4 denotes the running epochs for each method are equal to 10 to align the variables. Column 6 denotes the parameters of the learning model, and column 7 denotes the FLOPS (FLoating-point Operations Per Second) of each model. According to the result, DIComP are tens of thousands faster than previous works both in training speed and testing speed, and it has much fewer parameters and FLOPS of the model than the previous works. Because MLP can not benefit from GPU's parallel acceleration, otherwise the gap would even be wider. Even if we only have CPU, our method doesn't cost more than twice as much time. Column 8 denotes the model size, indicating DIComP can be easily packed as a quite lightweight tool for further binary analysis.

VII. Related Work

We summarize the related works and compare them with DIComP in terms of whether they identify the compiler (column 2), optimization level (column 3), and version (column 4), the technique (column 5), and efficiency (column 6) in Table XVII. Luca et al. [24] propose a structure-based representation for binary similarity and compiler inference. HIMALIA [23] and BinEye [25] leverage machine learning to mine the provenance of optimization levels. BinComp [22] and [21] infer the compiler, optimization level and compiler version with rule-based method and CRF (Conditional Random Field). Work [28] identify the compiler information with the help of CRF.

TABLE XVII: Comparison with related works.

	C.	O.L.	VER.	Tech.	Efficiency
i2v_RNN [24]	✓	✗	✗	RNN	✗
BinEye [25]	✗	✓	✗	CNN	✓
HIMALIA [23]	✗	✓	✗	Bi-GRU	✗
BinComp [22]	✓	✓	✓	✗	✗
Rosenblum2011 [21]	✓	✓	✓	CRF	✗
Rosenblum2010 [28]	✓	✗	✗	CRF	✗
DIComP (Ours)	✓	✓	✓	MLP	✓

However, none of them can meet all the needs. Even if the efficiency of BinEye is comparable to ours, but the work is limited by the hardware and needs to be accelerated with GPUs. DIComP infers all 5 common optimization levels of GCC and Clang with high accuracy, and comprehensively studies the binary appearance effected by the compilation options. What's more, we conduct a thorough study on how DIComP benefits other downstream missions, and it is really hardware friendly.

VIII. Conclusion

In this work, we thoroughly study how the different compilation options influence the appearance of the generated binaries. Based on the observations, we develop DIComP that leverages a lightweight neural network to infer the compilation options from stripped binaries. The comprehensive evaluations show that DIComP has remarkable speed and accuracy. We believe DIComP can benefit many downstream missions on analyzing binaries.

Acknowledgements

We sincerely thank the reviewers and anonymous shepherd for their valuable comments helping us to improve this work. This work was supported in part by grants from the Chinese National Natural Science Foundation (61272078, 62032010, 62172201, 61872180), Jiangsu "Shuang-Chuang" Program, and Jiangsu "Six-Talent-Peaks" Program.