Computer Security Symposium 2018
22 - 25 October 2018

コード断片からのコンパイラ推定手法

大坪 雄平[1,2]　大塚 玲[2]　三村 守[3,2]　榊　剛史[4]　受川　弘[1]　岩田　吉弘[1]

概要：同一のソースコードであっても，コンパイラの種類や最適化オプションの設定により，出力される実行コードの分布は大きく異なることが知られている．コンパイラの特徴は，「サブルーチン呼び出しの引数の指定方法」等特定の部分に大きく現れ，それ以外の部分はコンパイラ推定のノイズとなる可能性が考えられていた．過去に我々は，実行コード 16 命令の大きさのファイルの断片から実行コードか否かを 99%以上の精度で分類できる手法を提案した．この手法を活用し，コンパイラの種類および最適化オプションを組み合わせた 15 クラスの分類を行った．実験の結果，95%以上の精度で分類ができることが確認できた．したがって，コンパイラの差異により生じる実行コードの分布の差異は，16 命令でコンパイラ分類が行えるほど大きいことが明らかとなった．

キーワード：機械学習，CNN，バイナリ解析

Compiler Classification from a Code Fragment

Yuhei Otsubo[1,2]　Akira Otsuka[2]　Mamoru Mimura[3,2]　Takeshi Sakaki[4]　Hiroshi Ukegawa[1]
Yoshihiro Iwata[1]

Abstract: Even with the same source code, it is known that the distribution of outputted execution code greatly differs depending on the type of compiler and setting of optimization level. The feature of the compiler was thought to appear largely in specific parts such as “how to specify arguments of subroutine call”. Other parts were thought to be noise of compiler classification. In the past we have proposed a method which can classify whether or not it is executable code from a fragment of a file of size of executable code 16 instructions with accuracy of 99% or more. For this classification of 15 classes combining compiler type and optimization option, we conducted experiments on classification accuracy using this method. As a result, it was confirmed that classification can be performed with an accuracy of 95% or more. Therefore, we found that the differences in the distribution of execution code caused by different in compilers are so large that compiler classification is possible with only 16-opcode-sequence.

Keywords: Machine Learning, CNN, Binary Analysis

1. はじめに

マルウェアは 1 日あたり何十万個も新種が出ていると言われており，その全てを人が解析することは困難である．したがって，マルウェアを自動的に解析や分類できることが求められている．マルウェアを解析する手法は大きく分けて，動的解析と静的解析の 2 つに分けられる．それぞれの手法にはメリット・デメリットがあり，どちらの手法が優れているというわけではないが，大量のファイルを自動的に処理する場合は静的解析の方が必要なリソースが少ないことが多い．静的解析のうち実行コードを用いたマルウェアの検知や分類については，以前から様々な研究がされており [1][2][3]，その有効性が議論されている．

近年，畳み込みニューラルネットワーク（CNN）を始め

[1] 警察庁
National Police Agency
[2] 情報セキュリティ大学院大学
Institute of information security
[3] 防衛大学校
National Deffense Academy
[4] 東京大学
University of Tokyo

としたいわゆる深層学習を用いた研究が活発になっている．SVM や RF などの従来の特徴量ベースの手法は，特徴ベクトルの作り込みに重点が置かれていた．一方，深層学習においては，データに含まれる特徴は各層で自動的に学習するという発想から，特徴ベクトルの作り込みはあまり行わない．深層学習は様々な分野で優れた性能を発揮している．しかしながら，どのような特徴を学習しているかはブラックボックスであり，一見するとうまく機能しているように見えても，学習データによっては本来意図しない特徴を学習しているということがあり得る．

例えば，同一のソースコードであっても，コンパイラの種類や最適化オプションの設定により，出力される実行コードの分布は大きく異なることが知られている [3]．仮に実行コードを用いたマルウェアの検知や分類に深層学習を適用させた場合，どのような特徴を学習するかはブラックボックスであり，従来の手法よりもコンパイラの影響を強く受ける可能性がある．

過去の我々の研究で，深層学習を用いてファイルの断片から実行コードか否かを 99%以上の精度で認識できる手法（o-glasses）を提案した [4][5]．この精度は，入力データを x86/x64 実行コードとみなし，これを固定長命令に変換したものを 1d-CNN に入力することで，CNN の局所受容野と重み共有の仕組みを活用することで実現されている．しかしながら，学習用に実行コードとして準備したデータセットには，単一のコンパイラでコンパイルされたものしか含まれておらず，しかも最適化オプションの種類も単一であった．このため，他の種類コンパイラや最適化オプションで生成された実行コードを正しく実行コードとして認識するかは未知数であった．そこで本研究では，他のコンパイラや最適化オプションで生成された実行コードをデータセットに加えることにより，実行コードか否かの分類精度がどのように変化するか確認した．加えて，実行コードと判定した場合にコンパイラや最適化オプションの種類がどのくらいの精度で分類できるか確認した．

これまで，コンパイラの違いにより生じる差異は，「サブルーチン呼び出しの引数の指定方法」や「条件分岐命令の選択」等の特定部分以外はコンパイラ推定のノイズとなる可能性が考えられていた [6]．言い換えると，実行ファイルからランダムに取り出した小さなコード断片からはコンパイラを推定することは難しいと考えられていた．実験の結果，実行コードから位置を限定せずに取り出した 16 命令分のコード断片のみで，高い精度でコンパイラの推定ができることが確認できた．つまり，コンパイラの差異により生じる実行コードの差異は，従来考えられているよりも大きい可能性がある．

論文の貢献は次のとおりである．既存手法 o-glasses は，性能検証が実行コードおよび文書ファイルの 2 値分類しか行われておらず，適用できる領域が限定される可能性があった．本論文では，o-glasses の性能検証を行い，o-glasses がコンパイラ推定にも拡張可能であることを明らかにした．

論文の構成は次のとおりである．第 2 節では，本論文の関連研究を取り上げる．第 3 節では，既存手法 o-glasses について説明する．第 4 節では，コンパイラ推定に o-glasses を適用した結果について評価する．第 5 節では，考察を行い，第 6 節でまとめる．

2. 関連研究

2.1 コンパイラが実行コードに与える影響に関する研究

Walenstein らは，実行コードが類似していることにより発生するマルウェア解析上の問題点について述べている [7]．その中でいくつかの課題が取り上げられており，コンパイラや最適化オプションの影響の取扱や，プログラムにリンクされたライブラリの影響について述べられている．

岩村らは，実行コードの類似度の算出結果にコンパイラや最適化オプションの違いが影響を与えることを述べている [3]．同じソースコードであってもコンパイラが異なる場合，生成された実行コードの類似度は 5%から 10%の間という低い値となっている．

2.2 コンパイラおよび最適化オプションの推定に関する研究

碓井らは，シグネチャを用いたパターンマッチングにより，コンパイラを推定する手法を提案した [8]．シグネチャにはヘッダ等に含まれる様々なメタデータを活用している．しかしながら，これらのシグネチャの多くは，プログラムの動作に影響を与えないため，容易に書き換えが可能という特徴がある．マルウェアの中には，これらのメタデータを除去したり偽装したりするものがあることから，この手法は十分とは言えない．

また，碓井らは，隠れマルコフモデルによる最尤推定 [9] や，N-gram を用いた機械学習 [6][10] でコンパイラの最適化オプションを推定している．これらの手法は，統計処理を行うため，入力データにある程度の大きさを確保する必要がある．入力データとしては，実行ファイルのテキストセグメント全体を利用しており，これは実行コードがほとんどを占めている．しかしながら，テキストセグメントにはコンパイラの種類や最適化オプションの影響を受けない静的リンクライブラリのデータも混在している．したがって，前述した Walenstein らが指摘した，プログラムにリンクしたライブラリの影響という課題が残っている．

3. 1d-CNN による実行コードの認識

ここでは，我々が過去に提案した実行コードの認識に特化した学習器（o-glasses）について述べる．

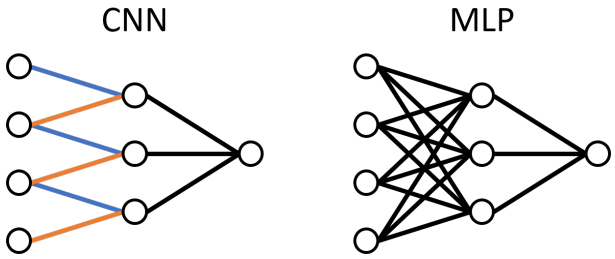

図 1　Schematic diagrams of a CNN and an MLP

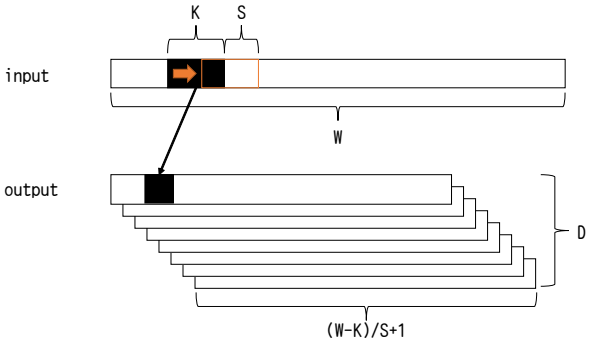

図 2　Illustration of the one-dimensional convolutional architecture

3.1　CNN

o-glasses は，CNN の局所受容野と重み共有という特徴に着目している．図 1 の左側は CNN の概念図を，右側は全結合（FC : Fully-Connected）の概念図を示している．図に示すとおり，全結合は各ノードがすべて接続しているのに対し，CNN は層ごとに接続の制限を行なっている．中間層は左側の層の一部の局所的なデータを入力としている．局所的なデータを入力としているが，次の右側の層でデータ全体を反映することが可能となっている．重み共有は，局所受容野のノードの重みを共有する仕組みである．図 1 であれば，青色の 3 本の線は同じ重みを共有している．赤色の 3 本の線も同じ重みを共有している．この局所受容野を利用することで，入力データの移動が発生しても，それぞれ移動普遍性を持った局所受容野の結果を用いることが可能となる．

いくつかのパラメータ（フィルタサイズ，フィルタの深さ，ストライドおよびゼロパディング）が畳み込み層の出力サイズを制御している（図 2）．フィルタサイズ（K）は，使用するフィルタ（局所受容野）の大きさである．フィルタの深さ（D）は，フィルタの種類のことで，この値がそのまま出力層の深さとなる．ストライド（S）は，フィルタを移動させる幅である．セロパディングは使用しないため，詳細は述べない．入力層の幅を W とすると，出力層の幅は $(W-K)/S+1$ で与えられる．

3.2　N ビット固定長命令用の局所受容野

o-glasses は，実行コードの認識に特化することを目指し

Program

```
0000:83EC14          SUB ESP, 0x14
0003:53              PUSH EBX
0004:55              PUSH EBP
0005:8B6C2428        MOV EBP, [ESP+0x28]
0009:8BC5            MOV EAX, EBP
000B:2D80000000      SUB EAX, 0x80
0010:742D            JZ 0x3F
0012:83E840          SUB EAX, 0x40
0015:741C            JZ 0x33
0017:83E840          SUB EAX, 0x40
001A:740B            JZ 0x27
001C:5D              POP EBP
001D:B8E0FFFFFF      MOV EAX, 0xFFFFFFE0
0022:5B              POP EBX
0023:83C414          ADD ESP, 0x14
0026:C3              RET
```

CFB

```
0000:D0CF            ROR BH, 0X1
0002:11E0            ADC EAX, ESP
0004:A1B11AE100      MOV EAX, [0XE11AB1]
0009:0000            ADD [EAX], AL
000B:0000            ADD [EAX], AL
000D:0000            ADD [EAX], AL
000F:0000            ADD [EAX], AL
0011:0000            ADD [EAX], AL
0013:0000            ADD [EAX], AL
0015:0000            ADD [EAX], AL
0017:003E            ADD [ESI], BH
0019:0003            ADD [EBX], AL
001B:00FE            ADD DH, BH
001D:FF09            DEC DWORD [ECX]
001F:0006            ADD [ESI], AL
0021:0000            ADD [EAX], AL
```

図 3　The results of disassembling a native-code or CFB (.doc) file

たネットワークモデルである．実行コードを 1 次元に並べたものを入力データとし，1d-CNN を用いることで，1 命令にだけ着目した局所受容野を形成する．これにより，その次以降の層では命令の関係性に着目した学習が期待される．実行コードおよび実行コード以外の例として CFB 文書ファイル（doc 拡張子，xls 拡張子，ppt 拡張子等）を逆アセンブルした結果を図 3 に示す．図に示すとおり，x86/x64 の命令の長さは様々である．しかも規則が複雑なため，命令を 1 byte ずつデコードしないと最終的な命令長は不明となっている．そのような実行コードをそのまま入力データとした場合，1 命令に着目した局所受容野の形成は行うことはできない．そこで，入力される実行コードを N ビットの固定長命令に変換し，フィルタサイズを N とする．さらに，ストライドも N とすることで，固定長命令の認識に特化した局所受容野の形成が期待される．

x86 の場合，Intel の仕様書をみると，基本的に命令の長さ 15 byte に制限されている [11]．しかしながら，文法上は 15 byte の制限を超えることができる．Intel の仕様書にも以下のように記述されている．

> Exceeding the instruction length limit of 15 bytes (this only can occur when redundant prefixes are placed before an instruction).

o-glasses では，念の為，仕様上の長さの制限を超えた命令も区別できるようにした 128-bit（16 byte）固定長としている．ただし，理論上の制限はない．

3.3　o-glasses

o-glasses のネットワーク全体を図 4 に示す．入力データは，2048 ビット値配列で，128-bit 固定長命令に変換された実行コードを 16 命令分としている．1 層目は 1d-CNN で，フィルタサイズを 128，ストライドを 128 とし，1 命令分の局所受容野を形成している．出力されるチャンネルの深さは，96 である．2 層目も 1d-CNN で，フィルタサイズを 2，ストライドを 1，チャンネルの深さを 256 としている．この層により，2 つの命令間の関係の特徴を得ることを期待している．3〜5 層目は全結合で，ノード数は 400，400，2 としている．また，3〜4 層目の入力部分には，ネッ

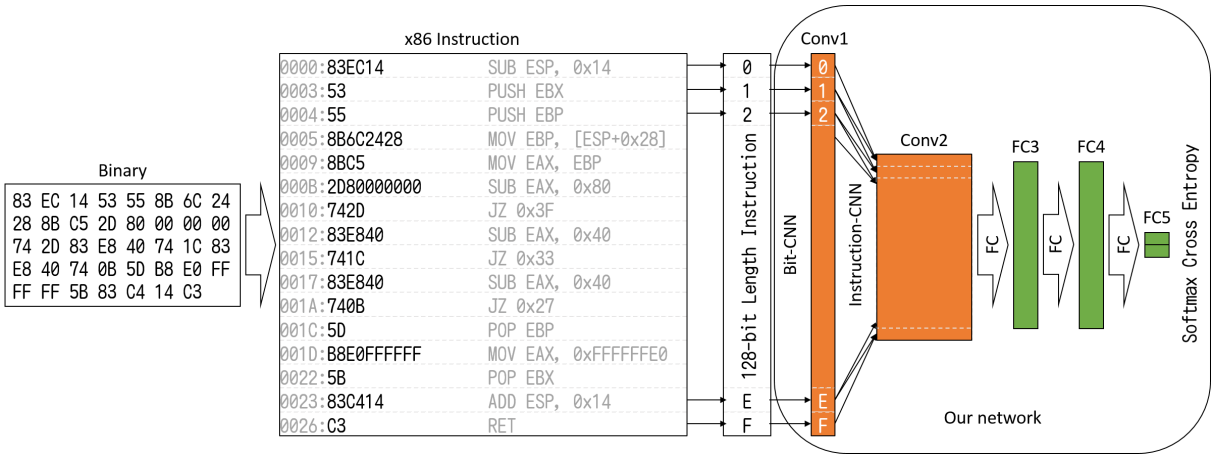

図 4　Outline of our 1d-CNN

トワークの学習プロセスをより安定化させ高速化するため，2015 年に Ioffe らが提案した Batch Nomalization[12] を取り入れている．中間層の活性化関数は，以下の式で示される ReLU[13] を使用している．

$$f(u) = \max(u, 0) \tag{1}$$

出力層の活性化関数は，Softmax 関数を使用している．ネットワークの出力層に分類したいクラス数 k と同数のノードを並べ，出力層 $l = L$ の各ユニット $k(= 1, \ldots, K)$ の総入力を $u_k^{(L)}$ で与えられたとき，Softmax 関数は以下の式で示される．

$$y_k \equiv z_k^{(L)} = \frac{\exp(u_k^{(L)})}{\sum_{j=1}^{K} \exp(u_j^{(L)})} \tag{2}$$

さらに，誤差関数は交差エントロピーを使用している．入力データを $x_n \in \{0,1\}^{2048}$，教師信号を $t_n \in \{0,1\}^K$，トレーニングサンプルの数を N，分類クラス数を K，モデルパラメータを $\mathbf{w}$ とすると，交差エントロピーは以下の式で示される．

$$E_n(\mathbf{w}) = -\sum_{k=1}^{K} t_{nk} \log y_k(x_n, \mathbf{w}) \tag{3}$$

3.4 SGD

この誤差関数を最小化するため，o-glasses では確率的勾配降下法（SGD: Stochastic Gradient Descent）を使用する．SGD とは，全訓練サンプル $n = 1, \ldots, N$ のうち一部の訓練サンプルを使い誤差関数 $E(\mathbf{w})$ を最小化するようにパラメータを更新する方法である．$E(\mathbf{w})$ は，各訓練サンプル 1 つだけについて計算される誤差 E_n の和として次の式で表される．

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \tag{4}$$

現在の重みを $\mathbf{w}^t$，学習後の重みを $\mathbf{w}^{t+1}$ とすると，パラメータの更新は次の式で表される．

$$\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \left. \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}=\mathbf{w}^t}, \tag{5}$$

ここで，η は，学習係数（learning rate）である．

o-glasses では，数値計算を効率化するため，ミニバッチ（minibatch）を利用する．これは，重みの更新を訓練サンプル 1 つ単位で行うのではなく，少数の訓練サンプルの集合をひとまとめにし，その単位で重みを更新する．m 番目のミニバッチを N^m とすると，全訓練サンプルの誤差 $E(\mathbf{w}^t)$ は次の式で表される．

$$E_m(\mathbf{w}^t) = \frac{1}{|N^m|} \sum_{n \in N^m} E_n(\mathbf{w}^t), \tag{6}$$

4. 実験

プログラムからランダムサンプリングしたコード断片にどの程度コンパイラの痕跡が残っているかを確認するため，o-glasses を用いて以下の実験を行なった．

4.1 実験データの準備

本研究では学習器を訓練させる段階で「どのような条件でコンパイルされたのか」わかっているサンプルが十分数必要である．そこで，インターネット上で入手できるオープンソースソフトウェアのソースコードを利用した．今回は GitHub[*1] 上でオープンソースソフトウェアを収集し，複数のコンパイラで最適化レベルを「なし」と「最大」に変えながらコンパイルを行った．コンパイルに成功したオブジェクトファイルのヘッダ情報を元に実行コード部分のみを取り出し，先頭から順番に 16 命令ずつ切り出すことでデータセットを作成した．今回準備したコンパイラは Visual C++ 2007（VC2007），Visual C++ 2003（VC2003），GCC および Clang の 4 種類である．VC2003 は 32bit のみでコンパイルし，それ以外は 64bit および 32bit の両方でコンパイルした．したがって，合計で 14 クラスのデータセットとなっている．

また，入力データが実行コードでない場合に対応するため，「Others」とラベル付けしたデータセットも作成した．

[*1] https://github.com/

これは，文書ファイルを元に作成した．文書ファイルは，文章，画像，メタデータなど様々な種類のデータが含まれている．その一方で，この文書ファイルがマルウェアでなければ，実行コードが含まれている可能性は極めて少ないと思われる．そこで，我々は検索エンジンを用いて「rtf」，「doc」，「docx」および「pdf」の 4 種類のファイルを収集し，VirusTotal[*2] を利用し，ウイルス対策ソフトでそのファイルがマルウェアとして検知されないことを確認した．その上で，文書ファイルを実行コードの集合とみなして，先頭から順番に強制的に逆アセンブルして 16 命令分ずつデータを切り出すことで，「Others」とラベル付けしたデータセットを作成した．

その結果．15 クラスで延べ 21,094 個のファイルから合計 4,987,513 個の訓練データができた．

4.2 コンパイラと最適化オプションの分類

以下の実験において，精度の測定は 10 分割交差検証を行い，その平均をとった．機械学習フレームワークとして利用したのは，Python 2.7.6 上の Chainer 4.0.0 である．学習係数は 0.01，ミニバッチサイズは 1,000，epoch 回数は 50 である．

データセットの概要および実験の結果を**表 1** に示す．表の「Compiler」はコンパイラの種類，「Arch.」は出力されるプログラムコードの対象 CPU アーキテクチャ（32bit および 64bit），「Opt.」は最適化オプションで「None」は最適化なしを「Max」は最大限の最適化を示す．「File」はコンパイルに成功したファイル数であり，そのファイルから作成した訓練データ数は「Sample」である．

「Precision」（適合率）は，分類結果の中にどの程度正解が含まれるかを示し，以下の式で示される．

$$Precision = \frac{TP}{TP + FP}, \tag{7}$$

「Recall」（再現率）は，正解のうちどの程度分類することができたかを示し，以下の式で示される．

$$Recall = \frac{TP}{TP + FN}, \tag{8}$$

ここで，TP は真陽性（True Positive），FP は偽陽性（False Positive），FN は偽陰性（False Negative），TN は真陰性（True Negative）を示す．

F-measure（F 値）は再現率と適合率の調和平均を取った値であり，以下の式で示される．

$$F_{measure} = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}. \tag{9}$$

F 値はある 1 つのクラスに着目した評価指標であり，今回の実験は多クラス分類であるため，評価指標として Macro-F1 および Micro-F1 も求めた．Macro-F1 は各クラスごとの F 値の単純平均であり，Micro-F1 は全体の TP，FN，FP および TN を和にしてから F 値を計算したものである．

[*2] https://www.virustotal.com/

5. 考察

5.1 コンパイラの種類増加の影響

実験の結果，「Others」部分に着目すると，実行コードか否かの分類精度は F 値で 0.9972 であった．我々が以前行った実行コードか否かの 2 値分類実験は，コンパイラの種類が VC2017 の 32bit で最大限の最適化の 1 種類のみで，F 値が 0.9990 であった．今回はコンパイラの種類が更に 13 種類増加して 15 クラス分類となっても，実行コードか否かの部分については，性能がほとんど低下していない．

このことから，コンパイラの種類が増加したとしても，実行コードか否かの分類精度にはほとんど影響を与えないことが分かった．このことから，未学習のコンパイラで作成された実行コードであっても，x86/x64 の実行コードであれば，コンパイラの推定結果が誤っていたとしても，実行コードか否かの分類は正しく行えることが期待される．

5.2 コンパイラの影響

これまで，コンパイラが出力する実行コードの差異として，「サブルーチン呼び出しの引数の指定方法」や「条件分岐命令の選択」などの特定部分以外はコンパイラ推定のノイズとなる可能性が考えられていた [6]．しかしながら，今回の実験の結果，実行コードから連続した 16 命令分のコード断片を抜き出すことができれば，位置に限定されず高い精度でコンパイラの推定ができることがわかった．言い換えると，16 命令分のコード断片であってもコンパイラの影響を強く受けることがわかった．この 16 命令は，一般的な関数よりも小さな実行コードサイズであるため，実行コードを用いた機械学習を行う場合は，コンパイラの影響を前提として学習用データを準備する必要がある．

5.3 静的リンクライブラリの影響

図 5 は，ある実行ファイルから取り出すコード断片の位置を先頭から 1 バイトずつスライドさせ，実験で用いた学習器で分類した結果を可視化したものを示している．

図の最も左側「Bit-Image」は，バイナリエディタ「Stirling」と同じ手法で実行ファイルを可視化したものである．Bit-Image は 1 ピクセルは 1 バイトに対応し，NULL（0x00）は白色，制御文字列（0x01〜0x1F）は水色，可読文字列（0x20〜0x7F）は赤色，その他は黒色が対応する．図の残りの 3 つの部分は，o-glasses で 15 クラス分類した結果を着色したものである．左側はコンパイラの種類に着目した着色であり，真ん中は Visual C++に着目した着色，右側は Visual C++に着目しつつバージョン違いについては焦点を当てない着色である．

表 1　データセット及び実験結果概要

Table 1　The overview of our dataset and the result of our experiment

Compiler	Arch.	Opt.	File	Sample	Precison	Recall	F measure
VC2017	32bit	None	1,170	369,605	0.8885	0.8824	0.8854
		Max	1,147	255,143	0.9196	0.9204	0.9200
	64bit	None	1,456	540,568	0.9935	0.9916	0.9925
		Max	1,242	542,020	0.9841	0.9813	0.9827
VC2003	32bit	None	1,350	292,277	0.8513	0.8534	0.8523
		Max	1,306	270,743	0.9236	0.9268	0.9252
	64bit	None	-	-	-	-	-
		Max	-	-	-	-	-
GCC	32bit	None	2,111	227,004	0.9793	0.9793	0.9793
		Max	1,844	239,821	0.9372	0.9459	0.9415
	64bit	None	1,582	283,276	0.9876	0.9858	0.9867
		Max	1,580	287,775	0.9245	0.9281	0.9263
Clang	32bit	None	1,205	101,024	0.9628	0.9664	0.9646
		Max	1,196	86,521	0.8533	0.8353	0.8442
	64bit	None	1,892	332,278	0.9893	0.9884	0.9888
		Max	1,883	246,500	0.9194	0.9086	0.9140
Others			130	912,958	0.9951	0.9994	0.9972
Total			21,094	4,987,513			
Macro					0.9406	0.9395	0.9401
Micro					0.9544	0.9543	0.9543

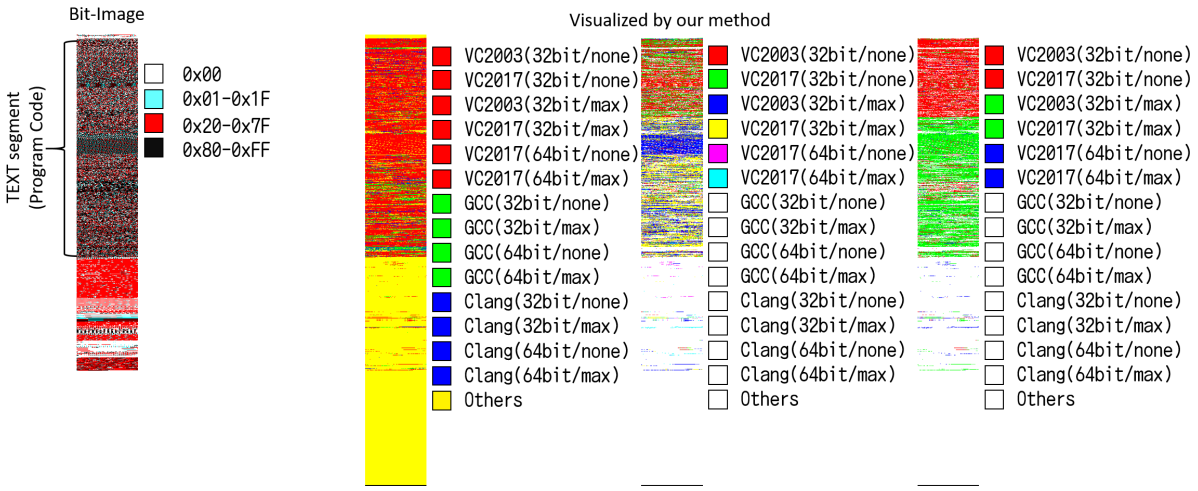

図 5　An executable file visualized by our method

図に示すとおり，実行コード部分の前半と後半で最適化オプションの推定結果が大きく異なる状況が観察された．前半部分は最適化なしで生成された実行コードであり，後半部分は最大限の最適化で生成された実行コードである．静的解析を行なったところ，前半部分が実行ファイルの作者が作成したと思われる実行コードであり，後半部分は静的リンクライブラリであった．リンクされる静的リンクライブラリの影響で，同一セクション中に複数の最適化オプションでコンパイルされた実行コードが混在していることが確認できた．加えて，o-glasses はその状況を上手く可視化することができた．

このことから，コンパイラや最適化オプションを推定する場合，同一セクション中に複数のコンパイラや最適化オプションが混在していることを前提としていない手法は，その精度に限界があることが推察できる．提案手法は，推定に必要な入力データが 16 命令分と小さいため，入力データには，単一のコンパイラおよび最適化オプションで生成された実行コードのみが含まれる可能性が高くなる．

5.4　o-glasses の限界

o-glasses は，実行コードの認識に特化した学習器であるが，単一の CPU アーキテクチャにのみ対応するという特徴を持つ．これは，入力データを固定長命令に変換する部分が CPU アーキテクチャ依存であるためである．本論文

で使用した o-glasses は x86/x64 の実行コードの認識に特化した学習器であるが，入力データを固定長命令に変換する部分を変更すれば ARM や MIPS 等の他の CPU アーキテクチャに対応することは可能である．しかしながら，2 つ以上の CPU アーキテクチャに同時に対応することはできない．

6. おわりに

本論文では，既存手法 o-glasses をコンパイラ推定に適用し，その性能評価を行った．その結果，16 命令分の実行コードから 95%の精度で最適化オプションを含めたコンパイラ推定をすることができた．このことから，o-glasses の適用領域がコンパイラ推定にも拡張できることが明らかになった．

今後は，o-glasses の適用可能領域について更に確認を行うとともに，静的解析の効率化に貢献できるように o-glasses の応用について研究を行う．

参考文献

[1] Weber, M., Schmid, M., Schatz, M. and Geyer, D.: A toolkit for detecting and analyzing malicious software, IEEE, p. 423 (2002).

[2] Li, W.-J., Wang, K., Stolfo, S. J. and Herzog, B.: Fileprints: Identifying file types by n-gram analysis, *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*, IEEE, pp. 64–71 (2005).

[3] Iwamura, M., Itoh, M. and Muraoka, Y.: Towards Efficient Analysis for Malware in the Wild, *2011 IEEE International Conference on Communications (ICC)*, pp. 1–6 (online), DOI: 10.1109/icc.2011.5963469 (2011).

[4] 大坪雄平， 大塚玲， 三村守， 榊剛史，後藤厚宏：1d-CNN による悪性文書ファイルに埋め込まれたプログラムコードの検知，暗号と情報セキュリティシンポジウム 2018 予稿集 (2018).

[5] Otsubo, Y., Otsuka, A., Mimura, M., Sakaki, T. and Goto, A.: o-glasses: Visualizing x86 Code from Binary Using a 1d-CNN, *arXiv preprint arXiv:1806.05328* (2018).

[6] 碓井利宣，松浦幹太：コンパイラ変更に対して頑強なマルウェア分類手法，コンピュータセキュリティシンポジウム 2014 論文集， Vol. 2014, No. 2, pp. 598–605 (2014).

[7] Walenstein, A. and Lakhotia, A.: The Software Similarity Problem in Malware Analysis, *Duplication, Redundancy, and Similarity in Software* (2006).

[8] 碓井利宣，松浦幹太：マルウェア対策技術の精度向上を目的としたコンパイラおよび最適化レベルの推定手法，コンピュータセキュリティシンポジウム 2013 論文集，Vol. 2013, No. 4, pp. 885–892 (2013).

[9] 碓井利宣，松浦幹太：機械語命令列の差異によるマルウェア対策技術への影響の削減を目的とした隠れマルコフモデルに基づくコンパイラ推定手法，暗号と情報セキュリティシンポジウム 2014 予稿集 (2014).

[10] 包含，碓井利宣，松浦幹太：特徴選択によるマルウェアの最適化レベル推定精度向上，暗号と情報セキュリティシンポジウム 2015 予稿集 (2015).

[11] Intel: Intel 64 and IA-32 architectures software developer manuals, `https://software.intel.com/en-us/articles/intel-sdm` (2016).

[12] Ioffe, S. and Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift, *arXiv preprint arXiv:1502.03167* (2015).

[13] Glorot, X., Bordes, A. and Bengio, Y.: Deep sparse rectifier neural networks, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 315–323 (2011).