実行ファイルの階層的構造に着目したマルウェア検知手法の検知能および判断根拠の検証

堀江, 健太 / HORIE, Kenta

(出版者 / Publisher)
法政大学大学院理工学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 理工学研究科編
(巻 / Volume)
64
(開始ページ / Start Page)
1
(終了ページ / End Page)
8
(発行年 / Year)
2023-03-24
(URL)
https://doi.org/10.15002/00026406

実行ファイルの階層的構造に着目したマルウェア検知手法の検知能および判断根拠の検証

EVALUATION OF PERFORMANCE AND INTERPRETABILITY OF A MALWARE DETECTION METHOD FOCUSING ON A HIERARCHICAL STRUCTURE OF EXECUTABLE FILES

堀江 健太
Kenta HORIE
指導教員 彌冨仁

法政大学大学院理工学研究科応用情報工学専攻修士課程

Machine learning-based malware detection methods realized fast, accurate, and flexible detection. However, their internal calculation processs tend to be black-boxed so they are unreliable and not used for commercial anti-malware softwares. Although some studies proposed interpretable malware detection methods to tackle to this problem, there still is a problem that there is no quantitative evaluation for the interpretability. In this study, we propose a novel method to evaluate the validity of the interpretability of a model by observing whether another malware detection method diagnoses a malware as a goodware after removing important components in it. Also, We propose a new interpretable malware detection method that utilizes a hierarchical structure in executable files: there are many functions in a file, and each function consists of assembly functions. Our model performs assembly instruction-level analysis, function-level analysis to analyze file using attention mechanism. By analyzing attention layers, our model can show which functions and instructions are suspicious. in this paper, We evaluated its performance and the validity of interpretability by experiments using applications of Android OS and Windows OS. As a result, we confirmed the model has high detection performance on both platforms, and the interpretability is verified on Windows applications.

Keywords *: Malware detection, Machine learning, Interpretability, Hierarchical analysis*

1. はじめに

情報通信機器の普及に伴い，新たに生み出されるマルウェアは急増しており，マルウェアを高速かつ高精度に解析することはサイバーセキュリティ分野における重要な課題である．近年数多く提案されている機械学習を用いたマルウェア検知手法は，亜種にも柔軟に対応でき，大量のマルウェアを高速かつ高精度にスキャンできる手法として期待されている．

一般に，マルウェア解析の方法は検体を実際に動作させ挙動を調べる動的解析と，検体を動作させずアセンブリコード等から挙動を調べる静的解析に大別される．動的解析ベースのマルウェア検知手法は難読化の影響を受けずに解析できるというメリットがあるが，サンドボックス検知を行うマルウェアの場合は悪意のある挙動を見せず正確な解析を行えない可能性があるというデメリットがある．その一方で，静的解析ベースの手法は難読化やパッキングされた検体に対しては弱いものの，動的解析ベースの手法よりも安全かつ網羅的に解析することができるメリットがある．

静的解析ベースのマルウェア検知手法としてはヘッダ情報を用いる手法 [1] や，命令の実行経路に沿って解析を行う手法 [2] などが存在するが，近年では実行ファイル中のコード部分は様々な関数で構成されており，さらに各関数は様々なアセンブリ命令から成り立っているという階層構造に着目し，アセンブリ命令を解析することで各関数を解析し，さらに各関数の解析結果をもとに検体全体を解析し識別を行うという階層的解析手法が提案されており，Windows [3, 4] や Android [5] のマルウェア検知において高い識別能を達成している．このような階層的解析は手動で行われる静的解析を模したリーズナブルな解析手法であり，静的解析ベースの手法の中でも特に高い識別能を達成することが期待できる．また，Windows や Android などのプラットフォームに関わらず，実行ファイルには命令，関数，検体という階層的構造が存在するため，モデル構造の観点ではプラットフォーム非依存な汎用性の高い手法であると考えられる．

これらの機械学習ベースのマルウェア検知手法は高い検知能や速度を達成しているにもかかわらず，現時点でアンチマルウェア製品として実社会に普及していない．これは，機械学習モデルの内部処理が black-box 化しやすいという特性により，手法に対する信用性が低いためであると考えられる．したがって，高い検知能に加え，判断根拠の解析が可能で信用できるマルウェア検知手法

が求められている．

この問題を解決するために，近年数多く提案されている機械学習モデルの判断根拠解析手法 [6, 7] や，判断根拠解析が可能な機械学習モデル [8, 9, 10] をマルウェア検知モデルに応用し，判断根拠解析が可能な信頼できるマルウェア検知手法を開発する試みがなされている [4, 11, 12]．これらの判断根拠の解析可能なマルウェア検知手法の提案により，機械学習を用いたマルウェア検知ソフトは実社会への応用および製品化の段階へと近づいていると考えられるが，判断根拠の妥当性が十分検証されていないという問題が残されていると考えられる．複数の先行研究 [4, 11, 12] では数個の検体において判断根拠となった部分を解析し，不審な通信を行う命令列があったなどの結果を挙げることで判断根拠の妥当性を主張したが，全ての検体を用いた網羅的な評価ができておらず，判断根拠の信頼性に疑問が残る．Wu らの研究 [13] では，既知のマルウェア解析レポートを用いることで判断根拠の妥当性を定量的に評価したが，この方法ではマルウェア解析の専門家によるマルウェア解析レポートを用意するコストがかかる．

そこで本研究では，まず判断根拠の妥当性の評価を低コストかつ網羅的に行える手法を提案する．この判断根拠の妥当性検証手法では，はじめに判断根拠の妥当性を評価したいマルウェア検知手法 M_{target} とは別に既に高いマルウェア検知能が確認されているマルウェア検知手法である“保証モデル”$M_{guarantee}$ を用意する．次に，マルウェアのみのデータセット X_{mal} 中の各検体について M_{target} が判断根拠とした部分を削除したデータセットと，そうでない部分を削除したデータセットを作り，$M_{guarantee}$ が前者のデータセット中の検体をより多くグッドウェアであると判定したならば，M_{target} が判断根拠とした部分は $M_{guarantee}$ にとっても有害，すなわち M_{target} の判断根拠は妥当であったとする．

また，本研究では，今後高い検知能が期待できる階層的解析手法 [3, 4, 5] と同じ階層的解析を行うが，どの関数が判断根拠になったかという関数レベルの判断根拠，各関数でどの命令が判断根拠となったかという命令レベルの判断根拠を提示することができる階層的解析手法を提案し，検知能を測定したほか，本研究で提案する判断根拠の妥当性検証手法を用いて関数レベルの判断根拠を定量的に評価した．さらに，階層的解析手法がプラットフォーム非依存で高い検知能および妥当な判断根拠提示能力を持つことを示すため，Windows アプリケーション，Android アプリケーションという 2 種類の環境で評価実験を行った．

2. 方法

(1) Windows アプリケーションにおける判断根拠解析が可能な階層的解析手法

(a) 手法の概要

検体をモデルに入力する前に，前処理として (b) 節に述べる方法で検体内に存在する N_{f} 個の関数を抽出する．その後，前処理後のデータをモデルに入力し，はじめに全てのアセンブリ命令のベクトル表現を求める処理が行われたのち，各アセンブリ命令のベクトル表現を用いて関数ごとにベクトル表現を求め，各関数のベクトル表現をもとに検体全体のベクトル表現を獲得し，検体がグッドウェアであるかマルウェアであるかの識別が行われる．

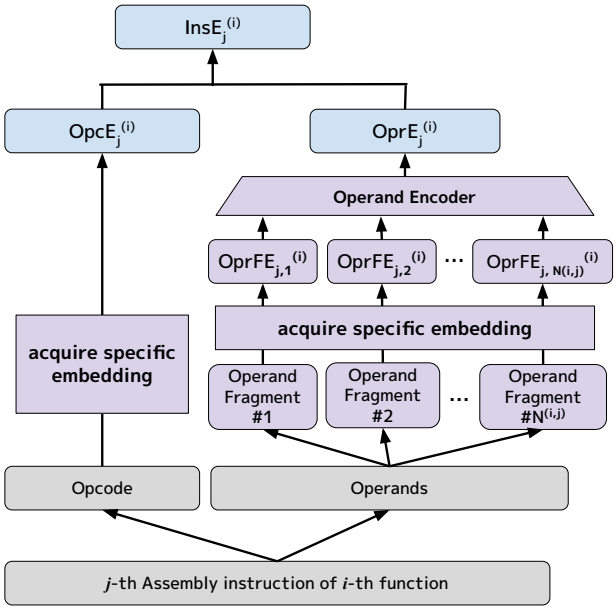

図 1　Windows アプリケーションに対する階層的解析モデルにおける，各アセンブリ命令のベクトル表現獲得を行う処理の概要図

各アセンブリ命令のベクトル表現が得られるまでの処理を図 1 に，各アセンブリ命令のベクトル表現が求められてから検体が識別されるまでの処理を図 2 に示す．まず，i 番目 $(i = 1, 2, \cdots, N_{\mathrm{f}})$ の関数内の j 番目 $(j = 1, 2, \cdots, N^{(i)})$ の命令について，(c) 節に述べる方法によりオペコードの固定長のベクトル表現 $\mathrm{OpcE}_j^{(i)}$ およびオペランドの固定長のベクトル表現 $\mathrm{OprE}_j^{(i)}$ を得たのち，各命令に対応する $\mathrm{OpcE}_j^{(i)}$ および $\mathrm{OprE}_j^{(i)}$ を結合することで命令の固定長のベクトル表現 $\mathrm{InsE}_j^{(i)}$ を獲得する．次に，関数ごとに全ての命令のベクトル表現 $\mathrm{InsE}_j^{(i)}$ の系列を Attention 機構付き双方向 LSTM および全結合層から構成される `Function Encoder` に入力することで，i 番目の関数に対応する固定長の関数のベクトル表現である $\mathrm{FuE}^{(i)}$ を得る．なお，`Function Encoder` における Attention 層を解析することで，関数の中でもどのアセンブリ命令が判断根拠となったのかを解析することができる．最後に，N_{f} 個の関数のベクトル表現 $\mathrm{FuE}^{(i)}$ の系列を Attention 機構付き双方向 LSTM および全結合層から構成される `File Encoder` に入力することで，検体の固定長のベクトル表現である `FiE` に変換し，`FiE` を全結合層に入力することで検体の識別を行う．なお，`File Encoder` における Attention 層を解析することで，どの関数が判断根拠となっているかを解析することができる．

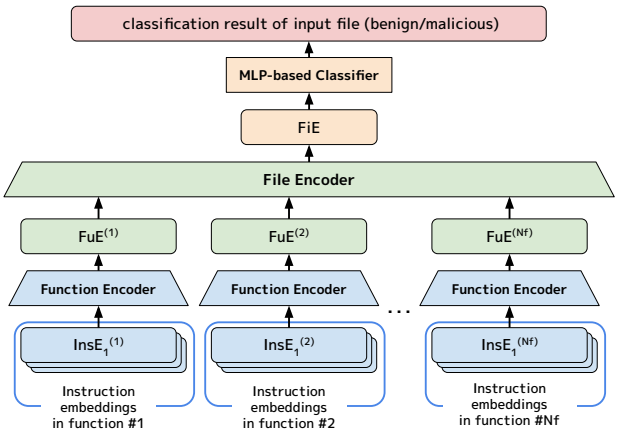

図 2　Windows アプリケーションに対する階層的解析モデルにおける，各アセンブリ命令のベクトル表現獲得以降の処理の概要図

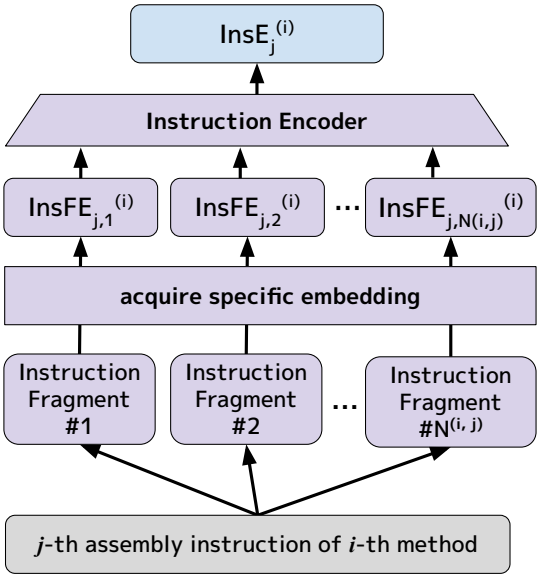

図 3　Android アプリケーションに対する階層的解析モデルにおける，各アセンブリ命令のベクトル表現獲得を行う処理の概要図

(b)　関数の抽出方法

一般に，x86 アーキテクチャでは関数の先頭には新たなスタックフレームを作成するための `push ebp` および `mov ebp,esp` という 2 つの命令が存在し，関数の末尾には呼び出し元の関数のスタックフレームに戻るための `ret` 命令が存在する [14]．このことを利用し，まず objdump[*1] を用いて検体中のコードが含まれるセクション内に含まれるアセンブリ命令を全て抽出し，その出力のうち `push ebp` 命令および `mov ebp,esp` 命令という 2 つの命令と，`ret` 命令の間に存在するアセンブリ命令を 1 つの関数内に含まれる命令列とした．

(c)　オペコードおよびオペランドのベクトル表現の獲得方法

図 1 に示したように，オペコードのベクトル表現 $\mathrm{OpcE}_j^{(i)}$ は，学習の経過に伴い更新される固定長のベクトル表現を与えることで獲得する．

オペランドのベクトル表現 $\mathrm{OprE}_j^{(i)}$ を得るには，はじめに Objdump の出力結果からオペランド部分の文字列を取得したのち，記号，演算子，数値，レジスタ名などのオペランドの構成要素 `Operand Fragment` に分解する．ここで，全ての数値は “<number>” タグとして扱う．これにより，例えばオペランド部分の文字列が “`edi, [ebp-0x88]`” の場合は，“edi”，“[”，“ebp”，“-”，“<number>”，“]” というような `Operand Fragment` に分解される．次に，k 番目 $(k = 1, 2, \cdots, N^{(i,j)})$ の `Operand Fragment` に対し学習の経過に伴い更新される固定長のベクトル表現 $\mathrm{OprFE}_{j,k}^{(i)}$ を与え，それらを Attention 機構付き双方向 LSTM および全結合層から構成される `Operand Encoder` に入力することにより獲得する．

(2)　Android アプリケーションにおける判断根拠解析が可能な階層的解析手法

(a)　手法の概要

検体をモデルに入力する前に，前処理として (b) 節に述べる方法で検体内に存在する N_m 個のメソッドを抽出する．その後，前処理後のデータをモデルに入力し，はじめに全てのアセンブリ命令のベクトル表現を求める処理が行われたのち，各アセンブリ命令のベクトル表現を用いてメソッドごとにベクトル表現を求める処理が行われる．次に，各メソッドのベクトル表現を用いて検体全体のベクトル表現を求め，検体がグッドウェアであるかマルウェアであるか識別結果を出力する．各アセンブリ命令のベクトル表現が得られるまでの処理を図 3 に示す．まず，i 番目 $(i = 1, 2, \cdots, N_\mathrm{m})$ のメソッド内の j 番目 $(j = 1, 2, \cdots, N^{(i)})$ の命令について，(c) 節に述べる方法により固定長のベクトル表現 $\mathrm{InsE}_j^{(i)}$ を獲得する．$\mathrm{InsE}_j^{(i)}$ の獲得から最終的な検体の識別までの処理過程に関しては，メソッドを関数と読み替えて (1) 節に述べた方法と全く同じ方法を適用する．すなわち，`Function Encoder` により各メソッドのベクトル表現を獲得し，`File Encoder` により検体全体のベクトル表現を獲得し，最終的な識別を行う．ここで，Windows アプリケーションに対する階層的解析手法と同様に，`Function Encoder` および `File Encoder` の Attention 層を解析することで，どの命令や関数が判断根拠となったのかを解析することができる．

(b)　メソッドの抽出方法

各メソッドに含まれる命令列を得るためには，はじめに検体に対し baksmali[*2] を適用することでアセンブリ命令が含まれる smali ファイルを抽出する．次に，メ

[*1] `https://sourceware.org/binutils/docs-2.26/binutils/objdump.html`

[*2] `https://github.com/JesusFreke/smali`

ソッド内の命令は method ディレクティブの範囲内，具体的には.method で始まる行と.end method の行に挟まれた部分に存在することから，.method で始まる行と.end method の行に挟まれた部分に存在する一連の命令をメソッドとし抽出した．

(c) アセンブリ命令のベクトル表現の獲得方法

Fig.3 に示したように，アセンブリ命令のベクトル表現 $\mathrm{InsE}_j^{(i)}$ は，はじめにアセンブリ命令の文字列を以下に述べるような方法で命令断片 Instruction Fragment に分解し，k 番目 $(k = 1, 2, \cdots, N^{(i,j)})$ の断片に学習の経過に伴い更新される固定長のベクトル表現 $\mathrm{InsFE}_{j,k}^{(i)}$ を与え，それらを Attention 機構付き双方向 LSTM および全結合層から構成される Instruction Encoder に入力することにより獲得する．検体から Instruction Fragment を得るまでの流れとしては，はじめに検体 APK ファイルに対し baksmali を適用することで smali ファイル群を獲得する．各アセンブリ命令の文字列について，オペコードとオペランドをひとまとめにし，空白を除去したのち，記号と通常文字列に分解する．その後，各命令断片について，レジスタのインデックスを除いた全ての数値を“<number>”タグに，ダブルクォーテーションで囲まれた文字列定数を“<string>”タグへと変換し，最終的な命令断片列を得る．例えば，命令“const-string /v0, "abc"”は“const”，“-”，“string”，“/”，“v0”，“,”，“<string>”というような Instruction Fragment に分解される．

(3) 判断根拠を解析可能なマルウェア検知モデルにおける判断根拠の妥当性の検証手法

関数，メソッドなどについて，入力のうちどの部分が判断根拠となったのかが解析可能であり，その判断根拠の妥当性を検証したい手法を M_{target} とする．まず，M_{target} と同じ特徴量を使用するものの異なるモデル構造を持ち，かつ既に高いマルウェア検知能力が確認されているマルウェア検知手法である“保証モデル”$M_{guarantee}$ を用意する．次に，データセット X 中のマルウェアの集合 X_{mal} 中の検体に対し M_{target} でスキャンを行い，i 番目の検体 $X_{mal}^{(i)}$ に対する判断根拠となった部分を解析する．次に，各検体 $X_{mal}^{(i)}$ について判断根拠となった部分を除去，すなわちマルウェアの無害化を行う．この判断根拠となった部分を除去する操作を important component removal (ICR) とし，ICR により無害化されたマルウェアの集合 X_{mal_icr} を構成する．それと同時に，判断根拠とならなかった非重要な部分を除去する操作を not-important component removal (nICR) とし，nICR を適用したマルウェアの集合 X_{mal_nicr} を構成する．X_{mal_nicr} の構成にあたっては，各検体における非重要な部分の選び方は無限に存在するため，乱数を用いて非重要な部分を選択するほか，様々な乱数シードを用いて複数の X_{mal_nicr} を構成する．i 番目の X_{mal_nicr} を $X_{mal\_nicr}^{(i)}$ とする．最後に，保証モデル $M_{guarantee}$ で X_{mal_icr} および全ての $X_{mal\_nicr}^{(i)}$ をスキャンし，全てのマルウェアのうちどれほどの割合を検知できたかを示す“悪性検出率”をデータセットごとに測定する．もし X_{mal_nifr} における悪性検出率がどの $X_{mal\_nicr}^{(i)}$ における悪性検出率よりも低くなれば，M_{target} が除去した要素は $M_{guarantee}$ から見ても有害であり，M_{target} の判断根拠は妥当であると評価する．

3. 評価実験

(1) データセット

各種評価実験にあたっては，(a) 節に述べる Windows アプリケーションのデータセット，(b) 節に述べる Android アプリケーションのデータセットをそれぞれ作成して使用した．いずれのデータセットもモデルの学習のための学習データ，学習終了基準を満たすか確かめるための検証データ，学習終了後に未知のデータに対する識別能を測定するためのテストデータで構成した．

(a) Windows アプリケーションのデータセット

学習データおよび検証データは，インターネット上で配布されていたデータセット[*3]から収集し，テストデータに関してはフリーウェアの配布サイト[*4] から得られたグッドウェア，MalShare [*5] から得られたマルウェアで構成されたデータセット[*6]から収集した．ただし，いずれのデータセットについても，任意の検体に対し様々なアンチマルウェアソフトでのスキャン結果を確認できる VirusTotal[*7]において 1 つ以上のアンチマルウェアソフトがマルウェアだと判断した検体をマルウェア，そうでない検体をグッドウェアであるとした．評価実験を行うにあたり，コード部分の大きい検体を用いるとメモリに乗り切らない場合があったことから，500 個未満の命令から成り立っている関数のみを関数として扱い，関数の個数が 300 個未満の検体のみを使用した．最終的な学習データ，検証データ，テストデータの検体数の構成を表 1 に示す．

表 1 Windows アプリケーション関連の評価実験で用いた各データセットの検体数の構成

データ種別	グッドウェア 検体数	マルウェア 検体数	合計
学習データ	500	500	1,000
検証データ	500	500	1,000
テストデータ	683	670	1,353

(b) Android アプリケーションのデータセット

学習データ，検証データ，テストデータのいずれも AndroZoo データセット [15] から構成した．AndroZoo データセットは長年にわたり APK ファイルの収集が続けられているデータセットであり，学習データおよび検証データは 2015 年に収集された検体，テストデータ

[*3] https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/
[*4] https://www.portablefreeware.com/
[*5] https://malshare.com/
[*6] https://www.reddit.com/r/datasets/comments/exhy38/malware_and_benign_windows_pe_cuckoo_reports/
[*7] https://www.virustotal.com/gui/home/search

は 2016 年に収集された検体から構成した．これらは同じ収集元ではあるが収集時期が異なるデータセット間では評価時の検知能が低下するということが報告されている [16] ことから，これらは別々の環境にあるデータセットとみなすことができると考えられる．Windows アプリケーションの場合と同様に，VirusTotal において 1 つ以上のアンチマルウェアソフトがマルウェアだと判断した検体をマルウェア，そうでない検体をグッドウェアであるとした．

評価実験を行うにあたり，コード部分の大きい検体ではメモリに乗り切らない場合があり，前処理に非常に長い時間がかかることから，メソッド中の命令数に関する制限は行わなかったものの，メソッドの個数が 300 個未満の検体のみを使用した．最終的な学習データ，検証データ，テストデータの検体数の構成を表 2 に示す．

表 2　Android アプリケーション関連の評価実験で用いた各データセットの検体数の構成

データ種別	グッドウェア検体数	マルウェア検体数	合計
学習データ	300	300	600
検証データ	100	100	200
テストデータ	200	200	400

(2)　評価実験

(a)　階層的解析手法の検知能の評価

本評価実験では，(1) 節に述べた Windows アプリケーションにおける階層的解析手法および (2) 節に述べた Android アプリケーションにおける階層的解析手法がそもそもマルウェア検知能を持つのかを確かめるため，(1) 節で作成したデータセットを用いて学習後のテストデータにおける検知能の測定を行った．検知能の評価にあたっては，各プラットフォームで高い検知能が報告されている既存手法との比較を行った．Windows アプリケーションに対する階層的解析手法の比較手法としては生データを入力として高い検知能が報告されている MalConv [17] を用い，Android アプリケーションに対する階層的解析手法の比較手法としては `classes.dex` の生データを入力し 1 次元 CNN 等を用いて解析を行う DexRay [18] を用いた．

評価指標としては，学習後のテストデータにおける正解率，適合率，再現率，F1 値，そして receiver operator characteristic (ROC) 曲線の下面積である area under the curve (AUC) を用いた．ROC 曲線は閾値を変化させたときの偽陽性率と再現率の関係を示す曲線であり，AUC は ROC 曲線と横軸に挟まれた部分の面積である．

(b)　階層的解析手法の判断根拠の妥当性の提案手法による検証

本評価実験では，(1) 節に述べた Windows アプリケーションにおける階層的解析手法および (2) 節に述べた Android アプリケーションにおける階層的解析手法のそれぞれについて，(3) 節に述べた方法により判断根拠の妥当性の検証を行った．

Windows アプリケーションにおける階層的解析手法ではどの命令や関数が判断根拠となったのかという判断根拠を，Android アプリケーションにおける階層的解析手法ではどの命令やメソッドが判断根拠となったのかという判断根拠を挙げることができるが，本論文では前者については関数レベルでの判断根拠を，後者についてはメソッドレベルでの判断根拠の妥当性を検証した．Windows アプリケーションにおける階層的解析手法の保証モデルとしては MalConv [17] を，Windows アプリケーションにおける階層的解析手法の保証モデルとしては DexRay [18] を利用した．関数およびメソッドレベルでの本研究で提案する判断根拠の妥当性の検証手法を適用するには，関数およびメソッドを除去する操作を定義する必要がある．Windows アプリケーションにおける関数の除去は，実行ファイル中の除去対象の関数内の命令部分の機械語をすべて `int3` 命令に対応する `0xcc` で置き換えることで実現した．`int3` 命令は `SIGTRAP` 例外を発生しプログラムを終了させる命令であり，Windows アプリケーションにおいては関数と関数の間に挿入されている命令である．この命令で関数部分を上書きすることにより，自然な関数の除去を行えると考えられる．Android アプリケーションにおけるメソッドの除去は次に述べるような方法で行った．はじめに APK ファイルに対し静的解析ツールである apktool[*8]を適用することで展開し，smali ファイル群を得る．次にその smali ファイル群から除去対象のメソッドを探し，除去対象のメソッド内に存在するアセンブリ命令を呼び出し元関数に戻る `return-void` 命令を除いて除去する．ただし，メソッドの中の `.prologue` ディレクティブ以前にあるメソッドの引数等を定義するディレクティブや命令については，削除すると次の手順である再構成を行えなくなるため残した．最後に，apktool を用いて改変済み smali ファイル群から APK ファイルを再構成することで検体内の対象メソッドの除去を完了する．

各手法の判断根拠に用いたマルウェアの集合 X_{mal} は，各プラットフォームのデータセットにおけるテストデータ中のマルウェアから構成した．ただし，Android アプリケーションのデータセット中のマルウェアは 200 体であったが，このうち 12 体は関数除去処理後の apktool による再構成がうまくいかなかったため，188 体を用いて評価を行った．X_{mal} から X_{mal_icr} および X_{mal_nicr} を構成するための ICR, nICR を行うにあたっては，削除する関数またはメソッドの個数を 1 個，2 個，全体の 1 割，全体の 2 割と変更し，各場合について悪性検出率の評価を行った．これらに加え，そもそも関数を抜くという操作が悪性検出率にどのような影響を及ぼすかも調べるため，関数を抜かなかった場合，すなわち 0 個の関数を除去した場合についても悪性検出率を評価した．また，nICR により m 個の関数またはメソッドを除去するにあたっては，重要度が上位 m 位までのもの以外から乱数によりランダムに m 個を選択し除去した．nICR を行うにあたってはどの関数を取り除くかで悪性

[*8] `https://ibotpeaches.github.io/Apktool/`

検出率が変化すると考えられるため，nICR に関しては 0, 1, 2, 3, 4 の 5 つのシード値で行い，ICR を行った場合とで悪性検出率に差があるかを評価した．乱数シード s による nICR を nICR(seed=s) と表記する．

4. 結果

(1) 階層的解析手法の検知能の評価結果

まず, Windows アプリケーションのデータセットにおいて階層的解析手法と MalConv の学習後に各種評価指標を測定した結果を表 3 に示す．表 3 から，本実験においては階層的解析手法のマルウェア検知能は MalConv のマルウェア検知能と同等以上の検知能を持っていることがわかる．

表 3　Windows アプリケーションにおける各手法の検知能の測定結果

手法	正解率	適合率	再現率	F1	AUC
階層的解析手法	0.775	0.734	0.855	0.790	0.833
MalConv [17]	0.724	0.725	0.724	0.724	0.803

次に，Android アプリケーションのデータセットにおいて階層的解析手法と DexRay の学習後に各種評価指標を測定した結果を表. 3 に示す．表 4 から，本実験においても階層的解析手法のマルウェア検知能は DexRay のマルウェア検知能とほぼ同等かそれ以上であることがわかる．

表 4　Android アプリケーションにおける各手法の検知能の測定結果

手法	正解率	適合率	再現率	F1	AUC
階層的解析手法	0.763	0.759	0.770	0.764	0.832
DexRay [18]	0.780	0.810	0.780	0.795	0.814

(2) 階層的解析手法の判断根拠の妥当性の提案手法による検証

はじめに，Windows アプリケーションのテストデータ中に含まれるマルウェアに対し，(b) 節で述べたように関数の除去数を変えながら，ICR および nICR を適用した各場合について保証モデル MalConv [17] で悪性検出率を測定した結果を表 5 に示す．表 5 より，いずれの関数の除去数の場合でも，ICR を行った場合の悪性検出率はどの nICR を行った場合の悪性検出率よりも低いことがわかる．また，ICR と nICR のいずれについても，1 個でも関数を除去した場合は関数を除去しなかった場合と比べ悪性検出率が 0.1 以上低下することがわかる．次に，Android アプリケーションのテストデータ中に含まれるマルウェアに対し，関数の除去数を変えながら，ICR および nICR を適用した各場合について保証モデル MalConv [17] で悪性検出率を測定した結果を表 5 に示す．表 5 より，いずれの関数の除去数の場合でも，ICR を行った場合の悪性検出率と nICR を行った場合の悪性検出率とで差は出ていないことがわかる．また，関数を除去した場合と除去しない場合と悪性検出率に差がなかったほか，表 4 における DexRay の適合率と関数を抜いていない場合の悪性検出率は同じくなるはずであるにもかかわらず，後者の方が悪性検出率が 0.1 程度高いことがわかる．

5. 考察

(1) 階層的解析手法の検知能の評価

Windows アプリケーションにおける評価実験では各種評価指標において階層的解析手法が既存手法 MalConv と同等の検知能を，Android アプリケーションにおける評価実験でも階層的解析手法は既存手法 DexRay と同等の検知能を示した．したがって，ハイパーパラメータのチューニングの余地はあると考えられるがいずれのプラットフォームにおいても階層的解析手法は既存手法と同等以上のマルウェア検知能を持つと考えられる．ARM などの他の命令セットやプラットフォームにおいても命令，関数，検体といった階層的階層を見出すことは可能であると考えられ，Android や Windows に限らず階層的解析手法はマルウェア検知手法として有効であることが期待できると考えられる．

また，モデル構造や使用する特徴量によってはさらに高い検知能を達成することが期待できると考えられる．モデル構造の観点では，今回は `File Encoder` や `Function Encoder` において系列データを処理するモデルとして Attention 機構付き双方向 LSTM を採用したが, I-MAD [4] で利用されていた Star-Transformer [10] や，各層の隠れ状態を独立に計算し勾配爆発および勾配消失を軽減し精度向上を達成した indRNN [19] など LSTM の上位互換となるモデルも多数提案されているため，これらや Attention 機構を利用することでさらなる検知能の向上が期待できると考えられる．使用する特徴量の観点では，第一に I-MAD のようにアセンブリ命令だけでなく，検体内の ascii 文字列情報や，インポートする関数の情報とも組み合わせることで検知能の向上を見込めると考えられる．また，今回評価を行った各階層的解析手法においては静的解析にあたって本来考慮する必要のある比較命令や条件付きジャンプ命令による制御構造を無視し，一連の命令を単純に系列データとして扱ったため，このグラフ構造を考慮できるようなモデルを応用することでさらなる検知能向上を見込めると考えられる．

(2) 階層的解析手法の判断根拠の妥当性の提案手法による検証

Windows アプリケーションに対する階層的解析においては, nICR のシード値や関数除去数に関わらず, ICR を行った場合の悪性検出率の方が nICR を行った場合の悪性検出率よりも低くなった．このことから，ICR で除去された関数は保証モデル MalConv [17] にとっても悪性のある関数であった可能性が高いと考えられる．さらに, MalConv は階層的解析手法とは異なりアセンブリ命

表 5　保証モデル MalConv [17] の各場合での悪性検出率

関数の除去方法	関数の除去数と悪性検出率				
	0 個	1 個	2 個	全体の 1 割	全体の 2 割
ICR	0.724	0.547	0.573	0.552	0.503
nICR(seed=0)		0.598	0.617	0.585	0.536
nICR(seed=1)		0.595	0.609	0.576	0.520
nICR(seed=2)		0.577	0.612	0.582	0.517
nICR(seed=3)		0.577	0.618	0.576	0.510
nICR(seed=4)		0.601	0.617	0.579	0.530

表 6　保証モデル DexRay [18] の各場合での悪性検出率

メソッドの除去方法	メソッドの除去数と悪性検出率				
	0 個	1 個	2 個	全体の 1 割	全体の 2 割
ICR	0.906	0.947	0.936	0.941	0.947
nICR(seed=0)		0.936	0.935	0.941	0.925
nICR(seed=1)		0.947	0.930	0.947	0.947
nICR(seed=2)		0.936	0.930	0.925	0.935
nICR(seed=3)		0.936	0.941	0.941	0.941
nICR(seed=4)		0.936	0.947	0.947	0.930

令を扱わず，ヘッダ情報や機械語などの生データを入力とし，モデル構造も異なることから，階層的解析手法の関数レベルでの判断根拠の妥当性は高いと考えられる．一方で，どの関数除去個数においても ICR と nICR での悪性検出率の差は 0.01〜0.04 程と小さい一方で，各検体でどの関数を除去するかという組み合わせの個数は非常に大きいため，ICR を行った場合の悪性検出率よりも nICR を行った場合の悪性検出率の方が低くなり，判断根拠が妥当であると評価できなくなる可能性もあると考えられる．したがって，本手法により判断根拠の妥当性を示しても信頼性が不十分であると評価される可能性がまだ残されており，より信頼性を高めるためにはより多くのシード値のもとで nICR を行い，保証モデルに関しても 1 つではなく複数用意すべきであると考えられる．

Android における階層的解析においては，nICR のシード値や関数除去数に関わらず，ICR を行った場合の悪性検出率と nICR を行った場合の悪性検出率に差がみられなかった．DexRay は階層的解析と同じ `classes.dex` を解析対象とすることや，検知能に関する実験で高い検知能が確認されていることから，本評価実験の結果を踏まえると Android における階層的解析の判断根拠の妥当性は不十分であると考えられる．しかし，命令，関数，検体全体という同じ階層的解析手法をとる Windows アプリケーションに対する階層的解析においては判断根拠の妥当性が確認されたことから，悪性検出率に差が出なかった原因は判断根拠が妥当ではない以外にある可能性も考えられる．例えば，本来ならば表 4 にあるデータセット中のマルウェア検知における適合率と，関数を抜かなかった場合の悪性検出率は等しくなるはずであるが，後者の方が悪性検出率が 0.1 程度高いという結果が出ている．データセット中のマルウェアと判断根拠の妥当性の検証に用いたマルウェアとでは本質的な差はないが，本評価実験を行うにあたっては，関数を抜かないマルウェアデータセットを作成するにあたっても apktool による展開と再構成を行った．これによるプログラムの本質的な変化はないと考えられるが，`classes.dex` の内容に若干変化が起こり，保証モデル DexRay の識別結果に影響を及ぼした可能性が考えられる．したがって，`DexRay` 以外にオペコードを解析対象とする保証モデルを用意することや，別の関数除去方法を適用することにより階層的解析手法の判断根拠の妥当性を認められる可能性が残されていると考えられる．

6. 結論

本研究では，まず判断根拠の妥当性を低コストかつ網羅的に検証する手法を提案したほか，どの関数が判断根拠になったかという関数レベルの判断根拠，各関数でどの命令が判断根拠となったかという命令レベルの判断根拠を提示することができる階層的解析手法を提案し，Windows アプリケーションおよび Android アプリケーションにおいて検知能を測定したほか，判断根拠の妥当性を検証した．その結果，いずれのプラットフォームにおいても階層的解析手法は既存手法と同等以上の検知能を持つことが確認できた．しかしながら，Windows アプリケーションにおける判断根拠の妥当性は保証することができた一方で，Android アプリケーションにおける判断根拠の妥当性は保証することができなかった．今後は，検知能と判断根拠の妥当性を併せ持つアンチマルウェア製品の実現に向け，検知能の向上を図ると同時に，保証モデルの増量や，関数の除去方法の改良などにより判断根拠の妥当性をさらに詳細に検証したい．

参考文献

[1] B. Kolosnjaji, G. Eraisha, G. Webster, A. Zarras, and C. Eckert, "Empowering convolutional networks for malware classification and analysis," in *Proc. International Joint Conference on Neural Networks*, pp. 3838–3845, 2017.

[2] S. Jeon and J. Moon, "Malware-detection method with a convolutional recurrent neural network using opcode sequences," *Information Sciences*, vol. 535, pp. 1–15, 2020.

[3] D. Gibert, C. Mateu, and J. Planes, "A hierarchical convolutional neural network for malware classification," in *Proc. International Joint Conference on Neural Networks*, pp. 1–8, 2019.

[4] M. Q. Li, B. C. Fung, P. Charland, and S. H. Ding, "I-mad: Interpretable malware detector using galaxy transformer," *Computers & Security*, vol. 108, p. 102371, 2021.

[5] J. Yan, Y. Qi, and Q. Rao, "Lstm-based hierarchical denoising network for android malware detection," *Security and Communication Networks*, vol. 2018, pp. 1–18, 2018.

[6] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Proc. Computer Vision and Pattern Recognition*, pp. 2921–2929, 2016.

[7] M. T. Ribeiro, S. Singh, and C. Guestrin, " "why should i trust you?" explaining the predictions of any classifier," in *Proc. International Conference on Knowledge Discovery and Data Mining*, pp. 1135–1144, 2016.

[8] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," in *Proc. International Conference on Machine Learning*, pp. 2048–2057, 2015.

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[10] Q. Guo, X. Qiu, P. Liu, Y. Shao, X. Xue, and Z. Zhang, "Star-transformer," in *Proc. Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 1315–1325, 2019.

[11] M. Kinkead, S. Millar, N. McLaughlin, and P. O' Kane, "Towards explainable cnns for android malware detection," *Procedia Computer Science*, vol. 184, pp. 959–965, 2021.

[12] H. Yakura, S. Shinozaki, R. Nishimura, Y. Oyama, and J. Sakuma, "Malware analysis of imaged binary samples by convolutional neural network with attention mechanism," in *Proc. Data and Application Security and Privacy*, pp. 127–134, 2018.

[13] B. Wu, S. Chen, C. Gao, L. Fan, Y. Liu, W. Wen, and M. R. Lyu, "Why an android app is classified as malware: Toward malware classification interpretation," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 2, pp. 1–29, 2021.

[14] "x86 アセンブリ言語での関数コール." `https://vanya.jp.net/os/x86call/`. Accessed: Jan. 30, 2023.

[15] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proc. International Conference on Mining Software Repositories*, pp. 468–471, 2016.

[16] A. Singh, A. Walenstein, and A. Lakhotia, "Tracking concept drift in malware families," in *Proc. of the 5th ACM Workshop on Security and Artificial Intelligence*, pp. 81–92, 2012.

[17] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole EXE," in *Proc. Workshops of the AAAI Conference on Artificial Intelligence*, pp. 268–276, 2018.

[18] N. Daoudi, J. Samhi, A. K. Kabore, K. Allix, T. F. Bissyandé, and J. Klein, "Dexray: a simple, yet effective deep learning approach to android malware detection based on image representation of bytecode," in *Proc. Deployable Machine Learning for Security Defense*, pp. 81–106, 2021.

[19] S. Li, W. Li, C. Cook, C. Zhu, and Y. Gao, "Independently recurrent neural network (indrnn): Building a longer and deeper rnn," in *Proc. Computer Vision and Pattern Recognition*, pp. 5457–5466, 2018.