情報処理学会論文誌 Vol.56 No.12 2289–2301 (Dec. 2015)

メソッドの呼び出し関係のグラフを用いたAndroidマルウェア検知手法の改良

庄田 祐樹[1] 金井 文宏[1,†1] 橋田 啓佑[1,a)] 吉岡 克成[2,3] 松本 勉[2,3]

受付日 2015年3月9日, 採録日 2015年9月2日

概要：Androidマルウェアの中には正規アプリに悪性コードを追加するリパッケージと呼ばれる手法で作成されるものがある．大量の正規アプリに対してリパッケージにより悪性コードが埋め込まれた例やリパッケージを自動で行うツールなどが報告されており，今後リパッケージマルウェアによる被害が増加すると予想される．リパッケージの特徴を利用したマルウェア検知手法として，元のアプリのコードと追加された悪性コードの関連性が薄いことに着目し，メソッドの呼び出し関係のグラフから，互いに呼び出し関係がないサブグラフを特定し，さらにマルウェアが用いることが多いAPIを含む割合によってマルウェア判定を行う手法が提案されている．本研究では上記の既存手法を改良して，メソッドではなくJAVAパッケージ単位でグラフを作成することで，より正確に悪性コードを区別する手法を提案する．提案手法では検知精度が向上するだけでなく，既存手法より小さい規模のグラフを用いることで，実行処理時間を大きく短縮することもできた．また，既存手法ではリパッケージによるコード挿入とよく似た特徴を持つ広告モジュールなどをホワイトリストで除外しているが，リパッケージマルウェアに特徴的な悪性コードの呼び出し方に着目することで，ホワイトリストを使わずに検知精度を高める方法を示す．

キーワード：Android，マルウェア，リパッケージ，モバイルセキュリティ

Improvement of Android Malware Detection Using Method Invocation Graph

Yuki Shoda[1] Fumihiro Kanei[1,†1] Keisuke Hashida[1,a)]
Katsunari Yoshioka[2,3] Tsutomu Matsumoto[2,3]

Received: March 9, 2015, Accepted: September 2, 2015

Abstract: Some of Android malware are made by technique called repackaging which adds malicious code to legitimate application. Security vendors reported a case in which malicious code was added to a large number of legitimate applications by automated repackaging. It is expected that the threats of repackaged application to increase in future. There is an existing technique for malware detection using features of the repackaged application with malicious code inserted, that is, there is a little relevance between added malicious codes and the codes of the original application. In the technique, an application is analyzed based on subgraphs that express invocations of methods used in the application. Then, each subgraph is tested if it contains APIs that are commonly seen in malware with high ratio. In this study, we improve the above existing technique. We propose a method to distinguish more accurately malicious code by creating a graph in JAVA package unit rather than a method and finding malicious sub-graphs in different approach from the existing technique. The proposed method shortened execution time by using small-scale graphs as well as improves the detection accuracy. Moreover, the existing technique depends on a white list to exclude legitimate advertisement modules and third-party libraries that have similar features to malicious code insertion by repackaging. We propose a method to distinguish between the legitimate modules from maliciously inserted code without white list by focusing on how candidate malicious code is called from the original application.

Keywords: Android, malware, repackaging, mobile security

1. はじめに

近年，Androidを狙ったマルウェアが増加し，問題となっている．Androidはスマートフォンの普及にともない，爆発的にシェアを拡大し，幅広い層の人々に利用されている．その中にはセキュリティやプライバシに関する知識に詳しい人やそういった意識の希薄な人までがAndroidのスマートフォンを所持している．一般のユーザが利用するスマートフォンには電話番号やメールアドレスなどの個人情報や固有のID，位置情報などプライバシに関わる様々な情報が含まれているが，そうした重大な情報が含まれているということを意識せずに利用しているユーザも少なくない．そうしたスマートフォンのOSとしての広いシェアに加え，Androidはオープンな開発環境や複数のアプリ配信サービスの存在などの理由から攻撃者に狙われやすい状況にある．実際にAndroidを狙ったマルウェアはここ数年で増加し続けている [1]．そのため，こうした脅威からユーザを保護するための技術が必要である．

Androidアプリはリバースエンジニアリングが容易であるという特徴があり，それを利用してリパッケージが行われることがある．リパッケージとはすでにパッケージ化されたアプリに対してリバースエンジニアリングを行い，その内容を改変したり，コードを追加したりする手法である．攻撃者はリパッケージにより，アプリの広告モジュールを改変して利益を得たり，悪性コードを追加しマルウェアを作成したりするなどの不正な活動を行っている．TrendMicroのレポート [2] によると，Google Playの無料の人気アプリ50個のうち77%ものアプリが無断で改変されてリパッケージアプリとして配信されていた．また，同レポートによるとリパッケージにより作成されたとされるアプリの半数以上がマルウェアであると報告されている．論文 [3] では収集したマルウェア1,260体のうち86%にあたる1,083体がリパッケージマルウェアであったとされている．

リパッケージマルウェアは元の正規アプリの機能も保持しているため，攻撃者は人気の正規アプリに対してリパッケージを行うことでユーザのダウンロードを促し，被害を拡大させていると考えられる．リパッケージマルウェアによる被害としてはAndroid.Troj.mdk[*1]の事例 [4] がある．このマルウェアは7,000種以上の正規アプリに対してリパッケージが行われ，中国のサードパーティマーケットで配布され最大で100万台のデバイスへ感染したとされる．また，AndroRATは大学のプロジェクトとしてオープンソースで開発されていたAndroid向けのRAT（Remote Access Tool）であるが，第3者がこのツールを悪用し，任意のアプリへAndroRATのコードをリパッケージにより追加するためのBinderと呼ばれるツールが売買され問題となった [5]．論文 [6] では収集した正規アプリに対してスクリプトによる自動リパッケージを行っている．その結果7割以上の正規アプリに対して自動リパッケージが成功しており，現状では正規アプリの多くがリパッケージに対する耐性を有していないことが示されている．

近年のこうした事例から，今後リパッケージマルウェアによる被害は増加すると予想される．リパッケージの特徴を用いたマルウェア検知の既存手法としては論文 [7] で提案されたMIGDroidという手法がある．これはリパッケージにより追加された悪性コードと元の正規アプリのコードの関連性が薄いことに着目し，メソッドの呼び出し関係のグラフから，互いに呼び出し関係がないサブグラフを特定し，さらにマルウェアが用いることが多いAPIを含む割合によってマルウェア判定を行う手法である．

しかし，元の正規アプリのコードから追加された悪性コードをメソッドにより呼び出している場合，元の正規アプリのコードと追加された悪性コードはメソッドの呼び出し関係のグラフ上でつながっており，同じサブグラフに属することがある．そのため，既存手法ではうまく検知できないリパッケージマルウェアが存在する．

そこで，我々はこの手法を基にしてJAVAパッケージ単位でグラフを作成し，既存手法とは異なる方法でサブグラフを作ることで，より正確に悪性コードを区別する手法を提案する．本手法により，既存手法と同様にマルウェアが用いることが多いAPIを含む割合によってマルウェアの検知を行い，既存手法よりも高い精度での検知に成功した．さらに，既存手法より大きい単位でグラフを作成することで，より小さい規模のグラフでの検知を行うことができ，結果として，既存手法よりも高速に処理を行うことができた．

また，広告モジュールやサードパーティ製のライブラリなどにはリパッケージの際のコード挿入とよく似た特徴を持つものがある．既存手法ではそうしたライブラリなどはホワイトリストにより，解析対象から除外している．本手法では，マルウェアが用いることが多いAPIの割合によって検知したサブグラフの呼び出され方に着目する．これにより，マルウェアによく見られる呼び出され方をして

1 横浜国立大学
Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
2 横浜国立大学大学院環境情報研究院
Graduate School of Environment and Information Sciences, Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
3 横浜国立大学先端科学高等研究院
Institute of Advanced Sciences, Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
†1 現在，NTTセキュアプラットフォーム研究所
Presently with NTT Secure Platform Laboratories.
本研究は横浜国立大学在籍時に行ったものである．
a) hashida-keisuke-dk@ynu.jp

[*1] Kingsoftによる検知名

いるサブグラフをよりマルウェアの可能性が高いものとしてBlack（悪性），そうでないものを広告モジュールなどの可能性を考えGray（悪性疑い）として検出する．Blackとして検出された検体をマルウェアとした場合，マルウェア検知率が70%のとき，提案手法の誤検知率を約11%から約2%まで下げることに成功した．

本論文では2章でAndroidに関する基本知識と研究背景について述べ，3章で既存手法について述べる．4章で今回実装した提案手法について説明し，5章で既存手法との比較を行う評価実験について説明し，6章で考察，7章でまとめと今後の課題について述べる．

2. 背景

2.1 Androidアプリの構造

Androidアプリは通常apkファイルと呼ばれるzip形式のファイルとして配布される．**図 1** にAndroidアプリのapkファイルの構造の例を示す．apkファイルの中には，アプリのJAVAの実行コードをdexという形式でまとめたclasses.dexファイル，アプリの動作に必要な情報を含んだAndroidManifest.xml，アイコンなどのリソースファイル，アプリの作成者の署名情報を含んだMETA-INFフォルダなどが含まれている．

Androidアプリを構成するコンポーネントは4種類ある．Activityはユーザアクティビティをともなうコンポーネントで，つねにフォアグラウンドで動作する．Serviceはユーザインタフェースを持たず，バックグラウンドで動作する．Broadcast Receiverはメッセージを受信して，動作を行うためのコンポーネントで，Content Providerはアプリ内のデータを他のアプリケーションと共有して利用するためのコンポーネントである．

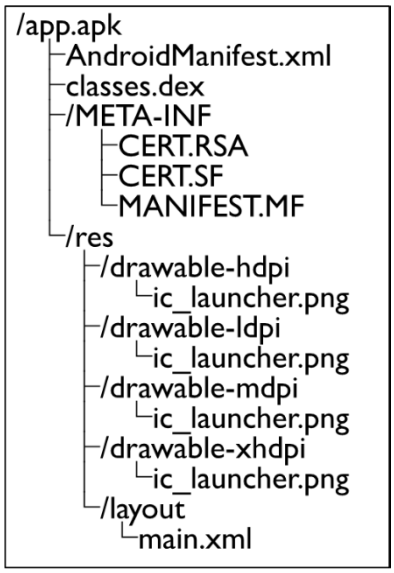

図 1 apkファイルの構成例
Fig. 1 Contents of an apk file.

これらのコンポーネント間の連携にはインテントが用いられる．インテントにはコンポーネントを指定して呼び出す明示的インテントとコンポーネントは指定せず，動作だけを記述する暗黙的インテントの2種類がある．暗黙的インテントではインテントフィルタにより動作を設定する．たとえば，ブラウザを開くという動作を行うインテントを起動した場合，android.intent.action.VIEWというインテントフィルタが発行される．このインテントフィルタを受け取る設定になっているアプリのコンポーネントがそれに対応して，起動する．インテントフィルタの設定はAndroidManifest.xml内に記述される．

アプリをホーム画面のアイコンから起動したとき，android.intent.action.MAINのインテントフィルタが発行される．起動するアプリのマニフェストファイル内でこのインテントフィルタを受け取る設定になっているアクティビティが最初に起動する画面になる．

2.2 リパッケージマルウェア

Androidの実行コードは一般的にJAVAで書かれており，リバースエンジニアリングが比較的容易である．apktool [8]やsmali/baksmali [9]といったツールを用いることで，アプリのコードを容易にsmaliと呼ばれる可読形式のファイルへ変換することができる．こういったツールを用いることでアプリのリパッケージが容易に行えることが論文 [6]で示されている．

今回はリパッケージを自動で行い，不特定多数のアプリへリパッケージによりマルウェアを作成する攻撃者を想定する．不特定多数のアプリへリパッケージを行う場合，確実に追加したコードが実行されるようにする場合コードの挿入位置方法は限られる．1つはマニフェストファイルを書き換え，アプリの起動時に最初に悪性コードが実行されるように設定する方法で，もう1つはアプリの起動時に呼び出される正規アプリのアクティビティから悪性コードを呼び出す方法である．また，悪性コードをインテントフィルタから呼び出す方法もある．上記のような方法で悪性コードを挿入しなくても，設定するインテントフィルタによっては一般的なユーザが高確率で行う動作によって悪性コードを呼び出すこともできる．

我々は実際のリパッケージマルウェアを静的解析し，それらが上記の手法にあてはまることを確認した．本論文では悪性コードがどのように呼び出されているかという観点でリパッケージマルウェアを以下の3種類に分類した．**図 2** は各タイプのマルウェアにおいて悪性コードの呼び出し方のフローの一例を示したものである．

1. 実行開始位置改竄タイプ
   このリパッケージではマニフェストファイルを書き換え，アプリ起動時に追加されたコード内にあるアクティビティが最初に起動するよう設定する．そのアク

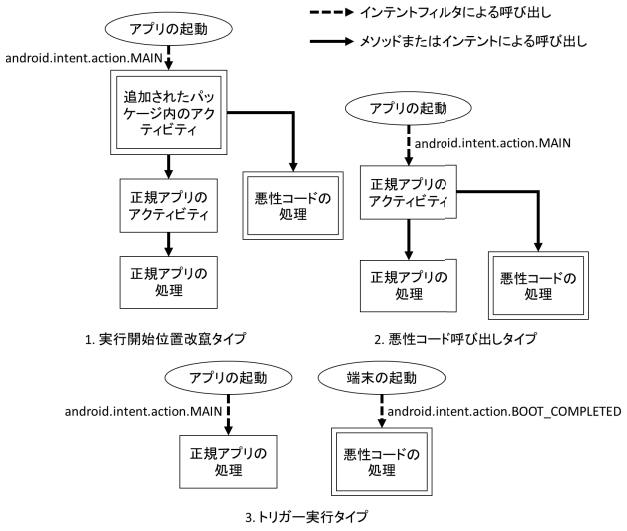

図 2 マルウェアの悪性コードが呼び出されるフロー例
Fig. 2 Insertion of malicious codes in each repackaging method.

ティビティは悪性コード本体を呼び出した後，正規アプリのアクティビティを起動し，自身のアクティビティは終了する．

このリパッケージでは悪性コードを追加し，マニフェストファイルの設定を書き換えるだけで行うことができる．マニフェストファイルへの改変は悪性コードの利用するパーミッションの追加，コンポーネントの追加と，アプリ起動時実行されるアクティビティの変更を行う必要がある．元の正規アプリの実行コードを改変する必要がないため，コードの追加は比較的容易に行うことができる．ただし，悪性コード内に呼び出す正規アプリのコンポーネント名を記述する必要があり，リパッケージごとに悪性コードを改変する必要がある．

このタイプのリパッケージマルウェアには Android.Geinimi [*2] や Android/DroidDream [*3] などがある．

2. 悪性コード呼び出しタイプ

このタイプのリパッケージでは元の正規アプリのアプリ起動時最初に呼び出されるアクティビティに悪性コードを呼び出すコードが書き加えられる．マニフェストの改変の内容は悪性コードの利用するパーミッションの追加とコンポーネントの追加のみである．アプリが起動した際，元の正規アプリのコードの中から悪性コードが呼び出されることになる．

このリパッケージでは，マニフェストファイルへの設定の追加に加え，正規アプリのコードを改変する必要がある．そのため，正規アプリが難読化されており，正規アプリのエントリポイントを見つけることができなければ，リパッケージに失敗する可能性がある．

このタイプのリパッケージマルウェアには Andr/Ksapp [*4] や Android.Troj.mdk などがある．

3. トリガ実行タイプ

このタイプのリパッケージでは上記 2 種のリパッケージと違い，起動後すぐには悪性コードが実行されない．悪性コードは特定のインテントフィルタを受け取るよう設定されており，なんらかのトリガにより悪性コードが起動する．

このリパッケージでは悪性コードの追加とマニフェストファイルへ設定の追加のみで行うことができる．マニフェストの改変の内容は悪性コードの利用するパーミッションの追加とコンポーネントの追加のみで，「実行開始位置改ざんタイプ」よりも改変は少ない．また，悪性コードも同じものを利用することができる．そのため，上記 2 種のリパッケージよりも成功率が高い．

このタイプのリパッケージマルウェアとしては Android/AndroRAT [*5] や Android/Dendroid [*6] がある．

3. 既存手法

3.1 Android におけるマルウェア検知手法

Android にマルウェアの検知手法は動的解析を用いるものと静的解析を用いるものがある．動的解析を用いた Android マルウェアの検知手法としては論文 [10] で提案されている CrowDroid や論文 [11] で提案されている AppsPlayGround などがある．これらはアプリの挙動から検知を行うものであり，未知のマルウェアも検知できる可能性がある．しかし，これらの手法ではマルウェアを実際に動かす必要がある．静的解析を用いた Android マルウェア検知手法としては論文 [12] で提案された DroidMat や論文 [13] で提案された手法がある．DroidMat はマニフェストファイルのパーミッションやコンポーネント API の呼び出しなどからマルウェア検知を行う手法であり，論文 [13] で提案された手法はアプリの制御フローを解析し，マルウェアの制御フローと比較し，マルウェア検知を行う手法である．これらは実際にマルウェアを動かす必要はないが，既存のマルウェアとの比較によりマルウェア検知を行っており，未知のマルウェアには対応できない．

3.2 MIGDroid

論文 [7] ではリパッケージマルウェアの特徴を利用して MIGDroid と呼ばれるマルウェア検知手法を提案している．リパッケージマルウェアは元の正規アプリのコードと追加された悪性コードの関連性が薄いことに着目し，メソッドの呼び出し関係のグラフ（MIG：Method Invocation Graph）から，互いに呼び出し関係がないサブグラフを特定し，さらにマルウェアが用いることが多い API を含む割合によっ

[*2] Symantec による検知名
[*3] F-Secure による検知名
[*4] Sophos による検知名
[*5] Fortinet による検知名
[*6] Fortinet による検知名

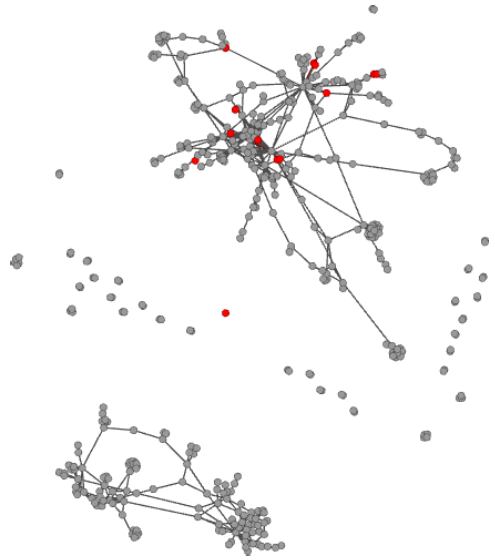

図 3 Android.Geinimi の Method Invocation Graph
Fig. 3 Method Invocation Graph of Android.Geinimi.

てマルウェア判定を行う手法である．

MIGDroid は静的解析により Android マルウェアの検知を行う手法であるが，マルウェアとのパターンマッチングによる検知ではなく，マルウェアが用いることが多い API がどの程度の割合で含まれているか，という指標でマルウェア判定を行う．そのため，未知のマルウェアも検知できる可能性がある．

図 3 は Android.Geinimi（MD5：B9E392DEE83D9596DD72B2739486062F）の Method Invocation Graph である．各ノードはメソッドを表しており，そのメソッド内から呼び出されているメソッドがエッジで結ばれている．マルウェアが用いることが多い API を呼び出しているメソッドは赤いノードで表している．この例では互いに呼び出し関係のない 2 つの大きなサブグラフがあるが，左下のサブグラフが元の正規アプリのコードであり，右上のサブグラフが追加された悪性コードである．Android.Geinimi は「実行開始位置改竄タイプ」のリパッケージマルウェアで，起動直後に呼び出された追加された悪性コードはインテントにより元の正規アプリのコードを呼び出すので，Method Invocation Graph では別々のサブグラフとなる．追加された悪性コードのサブグラフにはマルウェアが用いることが多い API が多く含まれており，元の正規アプリのコードには含まれていないことが分かる．

マルウェア判定のために，サブグラフのスコアを算出する．マルウェアがよく用いる API にはそれぞれスコアを割り振り，API のスコアリストを作成する．サブグラフ内のメソッドが呼び出している API を抽出し，その API がリストの中にあれば，その API のスコアを加算していく．同じ API が出てきた場合はそれぞれ別に加算している．サブグラフ内すべてのメソッドの API を抽出して算出された全メソッドのスコアをそのサブグラフ内にあるメソッド数で割ったものがそのサブグラフのスコアとなる．サブグラフのスコアは以下の式で表される．

$$\text{サブグラフのスコア} = \frac{\text{API のスコアの和 } (\in \text{API のスコアリスト} \cap \text{サブグラフ})}{\text{サブグラフに含まれる全メソッド数}}$$

サブグラフのスコアが閾値を超えた場合，その検体をマルウェアと判断する．

3.3 既存手法の問題点

既存手法の問題点としては以下の 3 つがあげられる．

問題点 1

MIGDroid ではメソッド単位でのグラフを作成しており，少数のメソッドで構成されるサブグラフが生成されることがある．MIGDroid ではサブグラフに含まれるマルウェアが用いることが多い API の割合でマルウェア検知を行っているため，サブグラフのメソッド数が少ないと含まれているマルウェアが用いることが多い API が少なくても割合が高いために誤検知されてしまう可能性がある．既存手法ではメソッド数が少ないないなどの特定の条件にあてはまるサブグラフはコード断片として検知対象から除外しているが，条件を厳しくすると見逃しが増えてしまう．

問題点 2

MIGDroid では 2 つのメソッド間で呼び出しがあればその時点で同じサブグラフに属してしまうため，元の正規アプリからメソッドにより悪性コードを呼び出している場合，元の正規アプリと悪性コードが同一のサブグラフに属することになるという問題点がある．実際に Andr/Ksapp や Android.Troj.mdk のような一部の「悪性コード呼び出し」タイプのリパッケージマルウェアは正規アプリから悪性コードをメソッドにより呼び出している．そのためサブグラフのスコアが小さくなり，マルウェアとして検知できない可能性がある．

問題点 3

ユーザに適した広告を表示するために個人情報を収集する広告モジュールやアクセス解析用のサードパーティ製のライブラリなどは，マルウェアが用いる API と同じ API を用いることが多く，誤検知の要因となってしまう．MIGDroid ではそうした広告モジュールやサードパーティ製のライブラリを，それらの JAVA パッケージ名をホワイトリスト化することで解析対象から除外している．しかし，広告モジュールやサードパーティ製のライブラリはいくつもの種類が存在し，それらすべてを網羅したホワイトリストを作成するのは困難であると思われる．また，単純な JAVA パッケージ名のホワイトリストでは，攻撃者が不正に改竄したライブラリを正規でない発行元から配布した場合などにそれらを検知できないという問題もある．

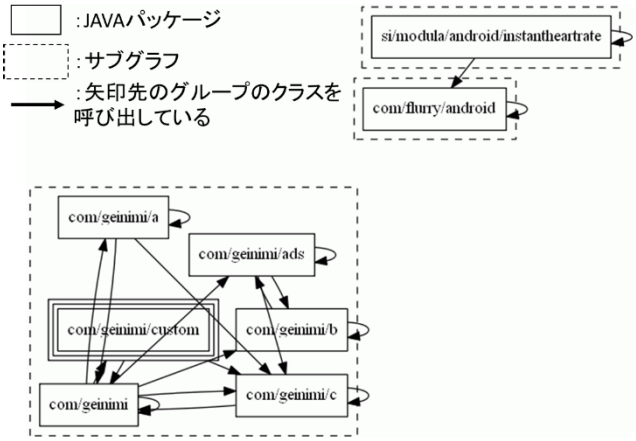

図 4 Android.Geinimi の JAVA パッケージ単位のグラフ
Fig. 4 Graph of JAVA packages of Android.Geinimi.

4. 提案手法

4.1 パッケージ単位でのグラフの作成

3.3 節の問題点 1 で述べたとおり，既存手法には少数のメソッドで構成されるサブグラフが生じるために誤検知を起こす可能性がある可能性がある．そこで JAVA パッケージ単位でのグラフを作成し，別の手法でサブグラフを作成することで，より正確に悪性コードを区別する．より大きな単位でグラフを作成することにより，コード断片の発生を減らすことができる．また，静的解析の結果から，リパッケージマルウェアにおいては悪性コードは JAVA パッケージ単位で追加されることが多いと考えられる．そのため，JAVA パッケージ単位でグラフを作成しても，悪性コード部分を切り分けるうえで問題はないと思われる．

4.2 サブグラフの作成

JAVA パッケージでグラフを作成した場合，そのまま既存手法の方法でサブグラフを作成すると，サブグラフが大きくなりすぎてしまう．また，3.3 節で指摘した問題点 2 である元の正規アプリと悪性コードは同じサブグラフに属してしまう問題もある．そこで，提案手法では既存手法とは異なった手法でサブグラフを作成する．JAVA パッケージを 1 つのノードとし，あるメソッドが別のパッケージのメソッドを呼び出している場合，それらをエッジとする．ここではエッジの向きも考慮している．次に各エッジに着目し，2 つの JAVA パッケージが当該エッジのみでつながっている場合は別のサブグラフに，当該エッジ以外にも他のエッジを介して互いにつながっている場合は同じサブグラフに属するとする．**図 4** は Android.Geinimi（MD5：B9E392DEE83D9596DD72B2739486062F）の JAVA パッケージ単位のグラフと作成したサブグラフの例である．

4.3 マルウェア判定

4.2 節の手法で作成したサブグラフの中に含まれている API のうち，マルウェアが用いることの多い API の割合により検知を行う．MIGDroid と同じ手法でサブグラフのスコアを付ける．そのサブグラフのスコアが閾値以上であれば，そのサブグラフを悪性コード候補として検出し，マルウェアの疑いがあると判断する．

この検知結果には広告モジュールやサードパーティ製のライブラリが含まれている可能性がある．そこで，検知されたアプリがマルウェアであるかどうかを悪性コード候補の呼び出し方により，Black（悪性）と Gray（悪性疑い）の 2 種類に分ける．

各タイプのリパッケージマルウェアの悪性コードはそれぞれ次のような呼び出され方をすると考えられる．

1. 実行開始位置改竄タイプ
   アプリの起動直後に悪性コードが呼び出される．
2. 悪性コード呼び出しタイプ
   アプリの起動直後に呼び出されるアクティビティからメソッド，もしくはインテントにより悪性コードが呼び出される．
3. トリガ実行タイプ
   悪性コードはインテントフィルタの設定によってなんらかのトリガにより呼び出される．

上記のようなコードの呼び出され方を検知対象とし，悪性コード候補が検知対象となる呼び出され方をしている場合は，その検体を Black（悪性），そうでない場合は Gray（悪性疑い）として検知する．

「実行開始位置改竄タイプ」は android.intent.action.MAIN のインテントフィルタを受け取り起動する場合を検知対象とする．「悪性コード呼び出しタイプ」については android.intent.action.MAIN のインテントフィルタを受け取るクラスの中の最初に実行されるメソッドである OnCreate()，OnStart()，OnResume() メソッド内からインテントにより呼び出されている場合のみを検知対象とした．「悪性コード呼び出しタイプ」には起動直後のアクティビティから同様にメソッドにより呼び出されている場合も含まれている．しかし，良性の広告モジュールやライブラリでも同様の呼び出され方をしており，誤検知が増える可能性があるため，対象としない．「トリガ実行タイプ」はマルウェアが用いるインテントフィルタとして，今回は**表 1** に示すインテントフィルタを用い，これらを受け取るクラスを含むサブグラフを検知対象とした．

本手法の全体の流れは**図 5** のようになる．アプリの実行

表 1 マルウェアがよく用いるインテントフィルタ
Table 1 Intent filters commonly seen in malware.

android.intent.action.BOOT_COMPLETED	端末の起動時に発行される
android.provider.Telephony.SMS_RECEIVED	端末がSMSを受信した際に発行される

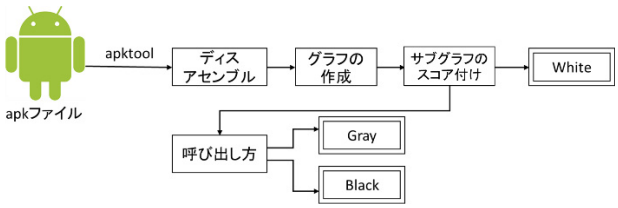

図 5 提案手法の流れ
Fig. 5 Flow of the proposed method.

コードを apktool でディスアセンブルし，生成された smali ファイルの中からメソッドの呼び出し命令である invoke-命令を抽出する．その後，4.2 節の手法で JAVA パッケージ単位のグラフを作成し，さらにサブグラフを作成する．このサブラフに対してスコア付けを行う．スコア付けの手法は既存手法と同じで，以下の式により求められる．

$$\text{サブグラフのスコア} = \frac{\text{API のスコアの和 } (\in \text{API のスコアリスト} \cap \text{サブグラフ})}{\text{サブグラフに含まれる全メソッド数}}$$

サブグラフのスコアが閾値を超えた場合，その検体をマルウェアと判断する．提案手法ではここまでをマルウェア検知部と呼ぶ．さらに呼び出し方を考慮し，閾値を超えた悪性コードがどのように呼び出されているかに着目し Black（悪性）と Gray（悪性疑い）の 2 種類に分ける．この部分を誤検知回避部と呼ぶ．以降ではマルウェア検知部を提案手法 1 とし，提案手法 1 の結果へ誤検知回避部を適用したものを提案手法 2 とする．

5. 評価実験

5.1 実験概要

今回，提案手法と既存手法の比較のために，MIGDroid の簡易版を論文 [7] を参考に実装した．実装した既存手法ではまず，アプリの実行コードを apktool によりディスアセンブルする．生成された smali ファイルから，各メソッドについて，メソッドの実行命令である invoke-命令を抽出し，呼び出されているメソッドをエッジで結んでいくことで Method Invocation Graph を作成した．サブグラフに対して，API によるスコア付けを行い，閾値以上であればマルウェアと判断する．ただし，サブグラフ内のメソッド数が 4 以下である場合はそのサブグラフはコード断片として検査対象から除外する．広告モジュールなどを検査対象から除外するためのホワイトリストは今回使用していない．

共同研究先のセキュリティベンダから提供していただいた実マルウェアを用いて提案手法の評価を行った．既存手法についても同じマルウェアセットに対して手法を適用し，検知精度を調べた．また，Google Play から収集した正規アプリ 517 体に対して，同様に手法を適用し，誤検知率を評価した．使用したマルウェアがよく利用する API のスコアリストは以下の方法で作成し，提案手法，既存手法ともに同じものを用いた．

表 2 マルウェアに含まれる API 上位 10 個
Table 2 Top 10 APIs used in malware.

API	スコア
SmsManager.sendTextMessage(String,String, PendingIntent,PendingIntent)	0.906886
SmsManager.getDefault()	0.906137
TelephonyManager.getLine1Number()	0.719417
TelephonyManager.getSubscriberId()	0.692065
SmsManager.getDefault()	0.692065
SmsManager.sendTextMessage(String,String, String,PendingIntent,PendingIntent)	0.688297
TelephonyManager.getSimSerialNumber()	0.637473
SmsMessage.createFromPdu(byte[])	0.621145
NetworkInfo.getExtraInfo()	0.608804
SmsMessage.createFromPdu(byte[])	0.604554

様々な種類のマルウェアが含まれるように選定した 1,288 体から API を抽出する．また，Google Play から収集した正規アプリ 517 体についても同様に API を抽出する．

各 API についてマルウェアに含まれる割合 X，正規アプリに含まれる割合 Y を算出する．

$$X = \frac{\text{AP が含まれていたマルウェア数}}{\text{マルウェアの全体数}}$$

$$Y = \frac{\text{API が含まれていた正規アプリ数}}{\text{正規アプリの全体数}}$$

次に，マルウェアに含まれ，正規アプリに含まれていない API を探す．X から Y を引いた値を Z とする．

$$Z = X - Y$$

Z の値が 0 以下のもの，また極小さい値のものは除外し，残った API のうち，個人情報の読み取りやファイルの読み書きなどマルウェアの挙動に直接関係のある API を抽出する．

ここで抽出された 485 個をマルウェアが用いることが多い API として今回の検知に用いる．API のスコアはこれら API に基礎点として 0.5 を与えた．さらに，よりマルウェアに多く含まれる API に重みを付けるために Z の値を加え，これを API のスコアとしても用いた．

$$\text{API のスコア} = 0.5 + Z$$

表 2 に今回用いた API のスコアが高い上位 10 個の API とそのスコアを示す．

5.2 検知精度の評価

検知の際のサブグラフのスコアの閾値を変更しながら，各手法をマルウェア 4,621 体へ適用した際の検知率（検知されたマルウェア/全マルウェア数* 100）と正規アプリ 517 体へ適用した際の誤検知率（検知された正規アプリ数/全正

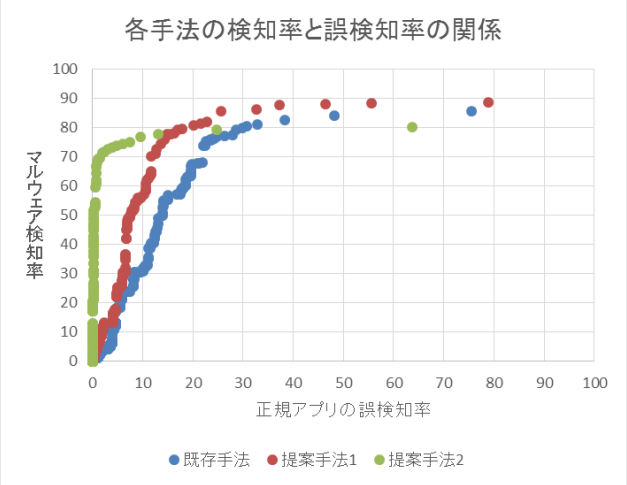

図 6 各手法の検知率と誤検知率の関係
Fig. 6 Relationship between malware detection rate and false positive rate in each method.

規アプリ数* 100）を調査した．閾値を 0 から 0.001 ずつ上昇させ，スコアの値が閾値を超えるサブグラフを含む検体をマルウェアと判断した．閾値を上げるにつれてマルウェア検知率，正規アプリの誤検知率はともに減少した．閾値が 0.538 以上のとき，いずれの手法においても正規アプリの誤検知率は 0%となり，閾値が 2.721 以上のとき，いずれの手法においてもマルウェアの検知率が 0%となった．

図 6 は各手法で閾値を変更しながら，それぞれの閾値におけるマルウェア検知率と正規アプリの誤検知率をプロットして作成したマルウェア検知率と正規アプリの誤検知率の関係を表すグラフである．この図 6 では縦軸がマルウェア検知率，横軸が正規アプリの誤検知率となっている．

既存手法と提案手法 1 を比較したとき，同じマルウェア検知率でも提案手法 1 のほうが少ない誤検知でマルウェアを検知できている．このことから，グラフの作成手法を変更したことで，検知精度が向上したということが分かる．

次に，提案手法 1 と提案手法 2 を比較する．提案手法 2 では提案手法 1 より検知できるマルウェアの数に限界があるものの，正規アプリの誤検知率がかなり低いことが分かる．なお，提案手法 2 では閾値が 0.073 以上のとき，正規アプリの誤検知率が 0%となった．

5.3 マルウェアファミリーごとの検知率

マルウェアファミリーごとの検知結果について調査をし，既存手法，提案手法でどのようなマルウェアが検知できてどのようなマルウェアが検知できないかを検証した．既存手法と提案手法 1 のそれぞれで正規アプリの検知率が 80%に近い値のときの結果を用いて，マルウェアファミリーごとの検知率についてまとめる．既存手法では閾値が 0.004 のとき，正規アプリの誤検知率が 30.75%でマルウェア検知率が 80.44%となった．提案手法 1 では閾値が 0.008 のとき，正規アプリの誤検知率が 20.12.%，マルウェア検知率が 80.81%となった．

検知名は F-Secure のものを用い，マルウェアの検知名からリパッケージにより作成されたと思われるマルウェアファミリーとリパッケージマルウェアではないと思われるマルウェアファミリーをセキュリティベンダのレポートなどから判断し，それぞれについて結果をまとめた．**表 3**-(I) はリパッケージマルウェアの結果で表 3-(II) はリパッケージではないマルウェアの結果である．

また，このときの提案手法 1 の結果へ呼び出し方を考慮することでどの程度検知率が低下するかを調べた．提案手法 1 のマルウェア検知率が 80%に近い値となる閾値 0.008 の際の提案手法 1 と提案手法 2 の検知結果を同様にまとめたものが**表 4**-(I) および表 4-(II) となる．

既存手法と提案手法で検知率が大きく変わったマルウェアファミリーについて詳細に解析した．

A) Andr/Ksapp

Andr/Ksapp は既存手法では検知できない可能性を指摘していた「悪性コード呼び出しタイプ」のリパッケージマルウェアである．実際の検知結果を確認すると，提案手法 1 ではいずれもすべての検体を検知できたのに対して，既存手法ではいくつかの検知漏れがあった．

Andr/Ksapp の検体のいくつかには com/simpleg という名前の JAVA パッケージが含まれており，これはリパッケージにより追加された悪性コードであると思われる．1 つの検体（md5：fe2bc4695852eef8ab3ce0e1d4570d050a47a3849da8ce9ecd4a80cf62f1904c）を例にすると，提案手法ではこの JAVA パッケージのみのサブグラフが存在し，そのスコアは 0.12087 であった．一方で，既存手法ではこの JAVA パッケージのほかに，リパッケージ元の正規アプリのコードである com/panda/sladr の一部が含まれており，スコアが 0.22159 まで下がっていた．

しかし，この com/simpleg は起動時最初のアクティビティからメソッドにより呼び出されており，これは広告モジュールなどと同様の呼び出し方であるため，提案手法 2 では他のマルウェアファミリーと比べて大きく検知率が下がってしまった．

B) Android.Spyware.GoneSixty

この検体に含まれる JAVA パッケージは com/gone60 という 1 つだけである．提案手法では JAVA パッケージ単位でサブグラフを作成しているため，この com/gone60 全体で 1 つのサブグラフと見なされてしまい，提案手法ではサブグラフのスコアが 0.00871 と小さくなってしまった．この検体を起動すると，com/gone60/MyService というクラスを起動し，個人情報を外部へ送信する．マルウェアが用いることが多い API を含む部分はこの com/gone60/MyService のみであり，他は UI や別の機能である．一方で，既存手法では JAVA パッケージより小さい単位で切り分けること

表 3 マルウェアファミリーごとの既存手法と提案手法 1 の検知結果の比較
Table 3 Detection results of each malware family by the existing method and the proposed method 1.

（Ⅰ） リパッケージマルウェア

	総数	既存手法	提案手法 1
Android/Geinimi	164 体	161 体	161 体
		98.17%	98.67%
Android/DroidKungFu	147 体	135 体	133 体
		91.83%	90.48%
Android/Pjapps	91 体	91 体	91 体
		100%	100%
Android/DroidDream	59 体	56 体	55 体
		94.92%	93.22%
Android/Bgserv	57 体	54 体	52 体
		94.74%	91.23%
Android/GoldDream	49 体	44 体	44 体
		89.80%	89.80%
Android/Adrd	23 体	22 体	22 体
		95.65%	95.65%
Android/Zsone	9 体	9 体	9 体
		100%	100%
Andr/Ksapp	12 体	10 体	12 体
		83.33%	100%
Android.Troj.mdk	15 体	9 体	15 体
		60%	100%

（Ⅱ） リパッケージではないマルウェア

	総数	既存手法	提案手法 1
Android/Boxer	464 体	464 体	464 体
		100%	100%
Android/Fakeinst	427 体	372 体	375 体
		87.12%	87.82%
Android/OpFake	376 体	329 体	371 体
		87.50%	98.67%
Android/Kmin	93 体	84 体	84 体
		90.32%	90.32%
Android/BaseBridge	70 体	48 体	62 体
		68.57%	88.57%
Android.Spyware.Gone Sixty	9 体	9 体	9 体
		100%	100%

表 4 マルウェアファミリーごとの提案手法 1 と提案手法 2 の検知結果の比較
Table 4 Detection results of each malware family by the proposed method 1 and the proposed method 2.

（Ⅰ） リパッケージマルウェア

	総数	提案手法 1	提案手法 2
Android/Geinimi	164 体	161 体	159 体
		98.67%	96.95%
Android/DroidKungFu	147 体	133 体	118 体
		90.48%	80.27%
Android/Pjapps	91 体	91 体	90 体
		100%	98.90%
Android/DroidDream	59 体	55 体	49 体
		93.22%	83.05%
Android/Bgserv	57 体	52 体	52 体
		91.23%	91.23%
Android/GoldDream	49 体	44 体	43 体
		89.80%	87.76%
Android/Adrd	23 体	22 体	22 体
		95.65%	95.65%
Android/Zsone	9 体	9 体	6 体
		100%	66.67%
Andr/Ksapp	12 体	12 体	6 体
		100%	50%
Android.Troj.mdk	15 体	15 体	5 体
		100%	33.33%

（Ⅱ） リパッケージではないマルウェア

	総数	提案手法 1	提案手法 2
Android/Boxer	464 体	464 体	461 体
		100%	99.35%
Android/Fakeinst	427 体	375 体	320 体
		87.82%	74.94%
Android/OpFake	376 体	371 体	368 体
		98.67%	97.87%
Android/Kmin	93 体	84 体	84 体
		90.32%	90.23%
Android/BaseBridge	70 体	62 体	40 体
		88.57%	57.14%
Android.Spyware.Gone Sixty	9 体	9 体	9 体
		100%	100%

ができ，サブグラフのスコアは 0.0348 と提案手法よりも高くなった．

5.4 実行処理速度の評価

提案手法では既存手法より大きい単位でグラフを作成しており，パフォーマンスの向上が期待される．既存手法と提案手法の実行にかかる処理時間の計測を行った．

マルウェア検知を行うスクリプトに処理時間を計測するための関数を埋め込み，正規アプリ 517 体とマルウェア 4,621 体に対して実行した．手法を適用する際にはまず，アプリを静的解析し，メソッドの呼び出し関係，API の呼び出しなどの情報をまとめたファイルを作成する．既存手

表 5 各手法の実行処理時間
Table 5 Processing time of each method.

(a) 正規アプリ 517 体への適用時

	既存手法	提案手法	
	グラフ + スコア	グラフ + スコア	インテント解析
全処理時間	10 時間 4 分 25 秒	44 分 19 秒	1 時間 38 分 21 秒
1 検体にかかる平均処理時間	70.15 秒	5.14 秒	11.42 秒
1 検体にかかる最大処理時間	1142.22 秒	553.91 秒	165.38 秒

(b) マルウェア 4621 体への適用時

	既存手法	提案手法	
	グラフ + スコア	グラフ + スコア	インテント解析
全処理時間	2 時間 3 分 18 秒	3 分 53 秒	39 分 5 秒
1 検体にかかる平均処理時間	1.60 秒	0.05 秒	0.51 秒
1 検体にかかる最大処理時間	193.31 秒	5.74 秒	32.15 秒

法，提案手法ともにこのファイルを入力とし，それぞれグラフの作成，サブグラフのスコア付けを行う．ただし，提案手法 2 を適用する際にはこれとは別にインテントによる呼び出し関係を解析する必要がある．今回計測を行ったのは既存手法の Method Invocation Graph を作成し，サブグラフのスコアを算出する部分と提案手法 1 の JAVA パッケージ単位でのグラフを作成し，サブグラフのスコアを算出する部分，インテントの呼び出し関係を解析する部分である．実験に用いたマシンは OS が Windows 7 Professional，CPU が Intel(R) Xeon(R) CPU E31225 @ 3.10 GHz，実装メモリが 16.0 GB のものを用いた．結果は**表 5** のようになった．

今回の実装では提案手法を適用する際にかかる時間は正規アプリ 517 体への適用時でグラフ作成，サブグラフのスコア付け，インテント解析すべてあわせると，2 時間 22 分 41 秒となり，既存手法の約 4 倍の実行処理速度となった．マルウェア 4,621 体への適用時では提案手法 2 を適用すると 42 分 58 秒となり，既存手法より約 3 倍高速に実行できることになる．呼び出し方による誤検知回避を行わない場合，提案手法 1 の処理速度は既存手法の正規アプリ 517 体へ適用したときで約 13 倍，マルウェア 4,621 体へ適用したときで約 32 倍とかなり高速に処理を行うことができる．これはグラフ作成の単位が大きくなったことでグラフ自体の規模が小さくなり，高速に処理できるようになったためである．

表 6 各手法のグラフの規模
Table 6 Size of the graph generated by each method.

(a) 正規アプリのグラフ

	既存手法	提案手法 1
1 検体の平均ノード数	12206.67	82.27
1 検体の平均エッジ数	15098.62	283.16
1 検体の平均サブグラフ数	6079.91	19.64

(b) マルウェアのグラフ

	既存手法	提案手法 1
1 検体の平均ノード数	1037.50	8.72
1 検体の平均エッジ数	1309.71	19.93
1 検体の平均サブグラフ数	514.21	3.54

既存手法の Method Invocation Graph と提案手法の JAVA パッケージ単位のグラフの規模を比較すると，**表 6** のようになる．既存手法におけるノード数はアプリのメソッド数であり，提案手法 1 におけるノード数は JAVA パッケージの数である．既存手法は無向グラフであるが，提案手法は有向グラフであるため，既存手法よりも提案手法のほうが，ノード数に対してエッジ数が多くなる．

6. 考察

6.1 マルウェアの検知精度について

既存手法と提案手法 1 を比較すると，全体として提案手法 1 のほうが少ない誤検知で多くのマルウェアを検知できるという結果となった．このことから，JAVA パッケージ単位でのグラフの作成で，メソッド単位でのグラフより正確に悪性コードを区別することができたといえる．特に，既存手法では検知できない可能性を指摘していた Andr/Ksapp や Android.Troj.mdk のような「悪性コード呼び出しタイプ」のリパッケージマルウェアの結果を見ると，実際に既存手法よりも提案手法 1 のほうが検知率が良かった．検知されたサブグラフを確認したところ，既存手法ではリパッケージにより追加された悪性コード以外の部分が多く含まれており，悪性コードをうまく識別できていなかった．JAVA パッケージ単位でのグラフが有効であった理由としては攻撃者が悪性コードをある程度モジュール化しており，それらが JAVA パッケージ単位で作成，追加されているということが推測される．攻撃者が悪性コードを追加するときと同じ単位でグラフを作成することで，悪性コードを識別しやすくなったと思われる．

しかし，Android.Spyware.GoneSixty のように既存手法よりも提案手法のほうが悪性コード部分のサブグラフのスコアが小さくなり，検知しにくいマルウェアも存在した．こうしたマルウェアについては，悪性コードが JAVA パッケージより小さい単位で埋め込まれていると考えられる．

検出した悪性コードがどのように呼び出されているかに着目することで，マルウェア検知の際の誤検知率を大きく

減らせるということを示した．リパッケージはソースコードがない状態で行われるため，コードの挿入方法はいくつかに限られ，広告モジュールなどとは異なった実装方法となることが多いと考えられる．そうした特徴を考慮することでホワイトリストを利用せずに誤検知を減らすことができた．しかし，Andr/Ksapp や Android.Troj.mdk のような一部の「悪性コード呼び出しタイプ」のリパッケージマルウェアは広告モジュールなどと同じ呼び出され方をしており，これらについては提案手法 2 により検知率が大きく下がってしまった．こうした「悪性コード呼び出しタイプ」のリパッケージマルウェアを検知するためには呼び出し方以外の特徴に着目する必要がある．

今回の評価実験ではホワイトリストを利用しなかったが，既存手法はホワイトリストを利用する前提での手法であり，ホワイトリストを利用した場合，検知精度は高くなると考えられる．しかし，既存手法を正規アプリへ適用した際に検知された検体の悪性コード部分と判断されたサブグラフを確認したところ，広告モジュールやサードパーティ製のライブラリ以外のコード部分も多く含まれており，ホワイトリストのみで誤検知を減らすには限界があると思われる．

6.2 実行処理速度について

JAVA パッケージ単位でグラフを作成することで，メソッド単位でグラフを作成するよりもグラフの規模が小さくなり，高速に処理を行えるようになった．ただし，正規アプリへの適用時の 1 検体にかかった平均処理時間と最大の処理時間を見ると，既存手法では平均で 70.15 秒，最大で 1,142.22 秒，提案手法では平均で 5.14 秒，最大で 553.91 秒なっており，提案手法は最大の処理時間が平均処理時間に比べて大きいことが分かる．また，マルウェアに対して各手法を適用したときのほうが正規アプリに対して各手法を適用したときよりも提案手法 1 の実行処理速度がより向上している．これはサブグラフを作成する際のアルゴリズムが既存手法に比べて提案手法のほうが複雑であるためであると考えられる．各手法のグラフの規模を見ると，マルウェアのグラフの規模は正規アプリのものよりも小さいことが分かる．グラフのノードやエッジの数が増えるにつれて，行う処理の数の増加率が既存手法よりも提案手法のほうが高いと考えられ，この増加率が大きくならないようなアルゴリズムの実装が必要である．また，静的解析を行う際はアプリの JAVA パッケージの種類や用いられるメソッドの種類などのアプリの複雑さが実行処理速度に影響するが，提案手法ではインテントの解析など，必要な解析が多く，複雑なアプリほどより処理に時間がかかると考えられる．

7. まとめと今後の課題

リパッケージの際の悪性コード挿入時の特徴に着目した Android マルウェア検知手法について提案した．JAVA パッケージ単位でグラフを作成することで，正確に悪性コードを区別し，マルウェアの検知精度を向上させることに成功した．さらに，既存手法よりも規模の小さなグラフを用いることで，処理にかかる時間を大きく短縮することができた．検出されたコードがどのように呼び出されているかを考慮することで，そのコードが悪性コードであるのか，広告モジュールなどであるのかを判別することができるということを示した．

今回は JAVA パッケージ単位で悪性コードを追加する攻撃者を想定していたが，JAVA パッケージより小さい単位で悪性コードが埋め込まれているため，提案手法では検知できないマルウェアが存在した．こうしたマルウェアは提案手法では悪性コード部分を区別することが難しく，今後，こうしたマルウェアを本手法でどのように扱っていくかを考えていく必要がある．

今後はより詳細な解析を行い，広告モジュールなどとよく似た特徴を持つ悪性コードを区別する方法を見つけることで検知精度を向上させる．また，本手法は JAVA で書かれた実行コードである dex ファイルのみを解析対象としているが，今後は C や C++ などによるネイティブライブラリや JavaScript で実装された悪性コードなどを検知する手法についても考える必要がある．

本手法は静的解析を用いたマルウェア検知手法であり，リフレクションによる呼び出しが行われた場合呼び出し先を特定することができないために正確なグラフを描くことができず，検知を行えない可能性がある．これは既存手法にも共通する課題である．リフレクションに呼び出し先を特定する手法としては論文 [14] や論文 [15] で提案されている手法があり，これらを併用し，呼び出し先を特定することでリフレクションを用いたマルウェアに対応することができる可能性がある．

今回検知に用いたマルウェアが用いることが多い API のスコアリストについては妥当性の評価を行っていない．この API のスコアリストは既存手法，提案手法ともに検知結果に大きく影響するものであるため，今後より適切な API のスコアリストの作成手法についても考えていく必要がある．

謝辞 本研究の一部は，総務省情報通信分野における研究開発委託/国際連携によるサイバー攻撃の予知技術の研究開発/サイバー攻撃情報とマルウェア実体の突合分析技術/類似判定に関する研究開発により行われた．また，本研究の一部は，文部科学省国立大学改革強化推進事業の支援を受けて行われた．

参考文献

[1] McAfee 脅威レポート：2014 年第 1 四半期 | McAfee セキュリティ研究レポート | セキュリティ情報 | マカフィー株式会社，入手先 ⟨http://www.mcafee.com/jp/threat-center/report/download86.aspx⟩ （参照 2014-12-08）.

[2] A Look at Repackaged Apps and their Effect on the Mobile Threat Landscape, available from ⟨http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-into-repackaged-apps-and-its-role-in-the-mobile-threat-landscape/⟩ (accessed 2014-12-08).

[3] Zhou, Y. and Jiang, X.: Dissecting Android malware: Characterization and evolution, *IEEE Symposium on Security and Privacy* (2012).

[4] MDK: The Largest Mobile Botnet in China | Symantec Connect Community, available from ⟨http://www.symantec.com/connect/blogs/mdk-largest-mobile-botnet-china⟩ (accessed 2014-09-12).

[5] 正規アプリが改造される恐ろしさ：IT&メディア：読売新聞（YOMIURI ONLINE），入手先 ⟨http://www.yomiuri.co.jp/it/security/goshinjyutsu/20131206-OYT8T00928.html⟩ （参照 2014-09-12）.

[6] 金井文宏，庄田祐樹，吉岡克成，松本　勉：Android アプリケーションの自動リパッケージに対する耐性評価，情報通信システムセキュリティ研究会（ICSS）(July 2014).

[7] Hu, W., Tao, J., Ma, X., Zhou, W., Zhao, S. and Han, T.: MIGDroid: Detecting APP-Repackaging Android-Malware via Method Invocation Graph, *The 23rd International Conference on Computer Communications and Networks* (*ICCCN 2014*) (2014).

[8] android-apktool – A tool for reverse engineering Android apk files – Google Project Hosting, available from ⟨http://code.google.com/p/android-apktool/⟩ (accessed 2012-12-12).

[9] smali – An assembler/disassembler for Android's dex format – Google Project Hosting, available from ⟨http://code.google.com/p/smali/⟩ (accessed 2012-12-12).

[10] Burguera, I., Zurutuza, U. and Nadjm-Tehrani, S.: Crowdroid: Behavior-Based Malware Detection System for Android, *SPSM'11*, October 17, 2011, Chicago, Illinois, USA (2011).

[11] Rastogi, V., Chen, Y. and Enck, W.: AppsPlayground: Automatic Security Analysis of Smartphone Applications, *CODASPY'13*, February 18–20, 2013, San Antonio, Texas, USA (2013).

[12] Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M. and Wu, K.-P.: DroidMat: Android Malware Detection through Manifest and API Calls Tracing, *ASIAJCIS '12, Proc. 2012 7th Asia Joint Conference on Information Security* (2012).

[13] 岩本一樹，西田雅太，和﨑克己：制御フローの比較による疑わしい Android アプリを絞り込む方法の提案，研究報告コンピュータセキュリティ（CSEC）(Dec. 2013).

[14] Bodden, E., Sewe, A., Sinschek, J., Oueslati, H. and Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders, *ICSE '11 Proc. 33rd International Conference on Software Engineering* (2011).

[15] Smaragdakis, Y., Kastrinis, G., Balatsouras, G. and Bravenboer, M.: *More Sound Static Handling of Java Reflection* (2014).

庄田 祐樹

2013 年 3 月横浜国立大学工学部電子情報工学科卒業．同年 4 月同大学大学院博士課程前期進学．Android マルウェアの解析等のモバイルセキュリティの研究に従事．2015 年 3 月横浜国立大学大学院環境情報学府情報メディア環境学専攻博士課程前期修了，修士（情報学）．

金井 文宏

2015 年 3 月横浜国立大学大学院環境情報学府情報メディア環境学専攻博士課程前期修了，修士（情報学）．同年 4 月より日本電信電話株式会社，NTT セキュアプラットフォーム研究所社員．同社サイバーセキュリティプロジェクト，セキュアネットワーク構成グループ所属．Android マルウェアの解析・検知やネットワーク攻撃の分析等，モバイル・ネットワークセキュリティの研究に従事．

橋田 啓佑

2014 年 3 月横浜国立大学工学部電子情報工学科卒業．同年 4 月同大学大学院環境情報学府情報メディア環境学専攻博士課程前期進学，在籍中．Android マルウェアの解析環境に関する研究等のモバイルネットワークセキュリティの研究に従事．

吉岡 克成 （正会員）

2005 年 3 月横浜国立大学大学院環境情報学府情報メディア環境学専攻博士課程後期修了，博士（工学）．同年 4 月独立行政法人情報通信研究機構研究員．2007 年 12 月より横浜国立大学学際プロジェクト研究センター特任教員（助教）．2011 年 4 月より横浜国立大学大学院環境情報研究院准教授．マルウェア解析やネットワーク攻撃観測・検知等のネットワークセキュリティの研究に従事．2009 年文部科学大臣表彰・科学技術賞（研究部門）受賞．

松本 勉 （正会員）

1986 年 3 月東京大学大学院工学系研究科電子工学専攻博士課程修了，工学博士．同年 4 月横浜国立大学講師．2001 年 4 月より同大学大学院環境情報研究院教授．2014 年 12 月より同大学先端科学高等研究院主任研究者を兼務．ネットワーク・ソフトウェア・ハードウェアセキュリティ，暗号，耐タンパ技術，生体認証，人工物メトリクス等の「情報・物理セキュリティ」の研究教育に 1981 年より従事．1982 年にオープンな学術的暗号研究を目指した「明るい暗号研究会」を 4 名で創設．2005～2010 年国際暗号学会 IACR 理事．1994 年第 32 回電子情報通信学会業績賞，2006 年第 5 回ドコモ・モバイル・サイエンス賞，2008 年第 4 回情報セキュリティ文化賞，2010 年文部科学大臣表彰・科学技術賞（研究部門）受賞．