Vol.2013-CSEC-63 No.2
2013/12/9

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

岩本 一樹[1,2,a)]　西田 雅太[1,b)]　和﨑 克己[2,c)]

概要：Android を搭載するスマートフォンの普及とともに Android のアプリも増加しており，Android の対象とするマルウェアも増加している．しかし公開されるすべてのアプリを解析者が解析することは困難なので，大量にあるアプリの中から解析者が解析するべき疑わしいアプリを絞り込む必要がある．本研究では Android アプリの制御フロー解析の結果から生成されたグラフを過去のマルウェアと比較することで，新たに公開されたアプリの中から疑わしいアプリを自動的に抽出する方法を検討する．同一の機能をもっているメソッドであっても，コンパイル環境やソースコードの機能に影響を与えないような違いによって，異なる命令列が生成される場合がある．ゆえに制御フロー解析の結果のグラフの不要なノードを削除することでグラフを縮約させ，命令列の違いに依らない特徴を抽出した．また，ノードの並び順を決めることで可能な限り隣接行列の比較だけで済むようにグラフを正規化することで，グラフの比較の高速化をはかった．結果，自動的に定義ファイルを作成し，効率よく解析するべきアプリを絞り込むことができた．

Proposal for Automated Method to Narrow Down to Suspicious Android Application by Comparing Control Flow Graph

Kazuki Iwamoto[1,2,a)]　Masata Nishida[1,b)]　Katsumi Wasaki[2,c)]

Abstract: The number of applications on Android has increased with the popularity of smartphones, and the number of malware on Android has also increased. However it is hard to analyze all of the applications, so we need to narrow down the huge amount of applications to only the suspicious applications which should be analyzed. In this paper, we propose the automated method to narrow down by comparing graphs which are generated by control flow analysis. There are cases where different instruction sequences which are generated from small difference in the source code or environment of the compiler, even though they have the same function. We extracted features which are independent from instruction sequence. And we have an increase in speed of comparison by using the canonicalizing graph we need only compare the adjacency matrix by determining the order of the node as much as possible. Finally, we could narrow down applications by using automatically generated definition files.

1. はじめに

モバイル端末を対象としたマルウェアは増加しており，その中でも Android で動作するマルウェアの占める割合は大きい [1]．また大量のアプリが公開されており，そのすべてを解析者が解析することは困難なので，大量にあるアプリの中から解析者が解析するべき疑わしいアプリを絞り込む必要がある．ゆえに本研究ではマルウェアに固有のメソッドの特徴を集めた定義ファイルを自動的に作成し，定義ファイルに含まれるメソッドがアプリに含まれる割合を求めることで，アプリのマルウェアらしさを包含度（Inclusion degree）として表すことで，解析するべき疑わしいアプリを絞り込む方法を提案する．

[1] 株式会社セキュアブレイン 先端技術研究所
Advanced Research Laboratory, SecureBrain Corporation, Chiyodaku, Tokyo, 102-0083
[2] 信州大学大学院総合工学系研究科
Interdisciplinary Graduate School of Science and Technology, Shinshu University, Matsumoto, Nagano, 390-8621
[a)] kazuki_iwamoto@securebrain.co.jp
[b)] masata_nishida@securebrain.co.jp
[c)] wasaki@cs.shinshu-u.ac.jp

アプリの動作は同じであってもコンパイル環境や僅かな改変によってバイトコードが異なる可能性がある．もし，同じ働きをする異なるバイトコードを同じ機能を持つと判別できなければ，似たようなコードを解析者が解析する状況が多くなる．ゆえにアプリから抽出される特徴はアプリの動作を反映していることが望ましい．

本研究ではアプリのメソッドを制御フロー解析した結果のグラフをメソッドの特徴とする．グラフを比較するにあたり，マルウェアから生成したグラフを Android のアプリがどれだけ含んでいるかを包含度で表す．あるアプリがマルウェアから生成したグラフを完全に含んでいるならば包含度は 100 となり，全く含んでいないならば包含度は 0 となる．この包含度と閾値を用いることで，一部のメソッドだけが改変されたマルウェアに対応する．包含度が大きい程，既知のマルウェアとバイトコードが類似しているので，そのアプリはマルウェアである可能性が高いと言える．

本研究では包含度に対して閾値を設定する．包含度が閾値よりも大きいならばマルウェアとみなし，包含度が閾値以下ならば通常のアプリとみなす．また包含度が閾値に近いアプリは誤って判別されている可能性が高いと考えられるので，閾値に近いアプリから順番に解析者が解析を行うことを検討する．

2. 本研究の提案

2.1 特徴抽出

本研究では Android のアプリに含まれる Dalvik VM の実行形式である classes.dex を baksmali[*1]で逆アセンブルし，メソッドごとに制御フロー解析を行いグラフを生成する．グラフのノードのラベルは対応するバイトコードで呼び出しているメソッドの名前になる．グラフのエッジはあるメソッドが呼ばれた後に呼ばれる可能性があるメソッドを表している．2 つ以上のメソッドが連続して呼ばれる場合には，複数のメソッドを 1 つのノードにまとめる．ただしアプリ内部のメソッドを呼び出す場合には，呼び出すメソッドの名前はアプリの作成時に任意に決めることができる．そのため同一のメソッドを呼び出していても異なる名前になる可能性があるので，この場合は「*」をメソッドの名前の代わりに用いることで抽象化する．またログの記録などのアプリの機能には関係ないメソッドの呼び出しは削除する．

たとえば**表 1** はあるマルウェアのメソッドを JEB[*2]で逆コンパイルした結果である．表 1 のメソッドを制御フロー解析したグラフは**図 1** になる．

2.2 グラフ比較

本研究ではグラフの隣接行列を比較することでグラフの

[*1] http://code.google.com/p/smali/
[*2] http://www.android-decompiler.com/

表 1 逆コンパイルされたメソッド
Table 1 Decompiled Method

```
public boolean canwe() {
    boolean v0;
    String v12 = "was";
    String v9 = "no";
    Cursor v8 = this.db.query("table1",
        new String[]{v12},
        null, null, null, null, null);
    if(v8.moveToFirst()) {
        do {
            v9 = v8.getString(0);
            if(v8.moveToNext()) {
                continue;
            }
            break;
        }
        while(true);
    }
    if(v8 != null && !v8.isClosed()) {
        v8.close();
    }
    if(v9.equals(v12)) {
        v0 = false;
    }
    else {
        v0 = true;
    }
    return v0;
}
```

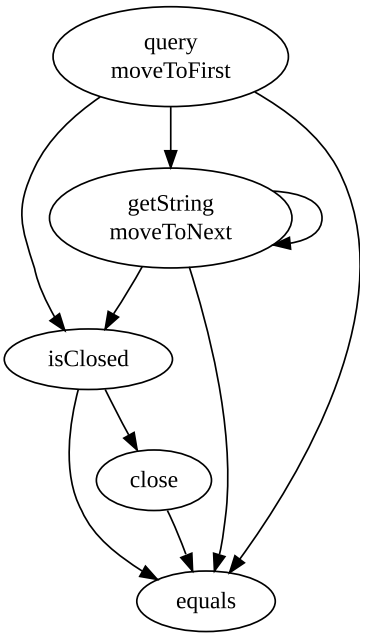

図 1 制御フローグラフ
Fig. 1 Control Flow Graph

比較を行う．たとえば図 1 の隣接行列は

$$A_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

である．しかし隣接行列のノードに対応する行と列を入れ替えても同じグラフを表現できるので，隣接行列は A_1 だけではない．ゆえに隣接行列の要素を比較するだけではグラフを比較することはできない．

そこで本研究ではノードの順番を次のように決定することで，隣接行列とノードのラベルを比較するだけでグラフの比較を行えるようにする．

(1) 単純な比較
  (a) ノードのラベルの辞書順
  (b) ラベルが同じときには入力エッジの数が少ない順
  (c) 入力エッジの数も同じときには出力エッジの数が少ない順
  (d) 自身へのエッジがあるときには順位が後ろになる
(2) 他のノードとの関係による比較
  (a) 順位が確定したノードからの入力エッジの有無
  (b) 順位が確定したノードへの出力エッジの有無
  (c) 順位が前のグループのノードに対して入力エッジの数が少ない順
  (d) 順位が後のグループのノードに対して入力エッジの数が少ない順
  (e) 順位が前のグループのノードに対して出力エッジの数が少ない順
  (f) 順位が後のグループのノードに対して出力エッジの数が少ない順
(3) 到達による比較
  (a) ノードから到達できるノードの数が少ない順
  (b) ノードへ到達できるノードの数が少ない順
(4) 同じ形になるノードが複数ある場合の順番確定

たとえば**図 2** では端にあるラベルが a，c，d，e のノードは（1）で順番が確定するが，（2）でなければ中間にあるラベルが b のノードの順番を決めることができない．**図 3** の左の 2 つのループ状のノードと右の 3 つのループ状のノードは（3）による到達できるノード数以外に違いはない．また**図 4** のラベルが b と c のノードは対称的であり，いずれかの順番を決めなければもう一方の順番も決めることができないので，（4）で決定する．

しかし（4）までの処理を行ってもループ状のグラフは隣接行列が一意に決まらない可能性がある．たとえば図 3 の右の 3 つのノードは一意に決まらず，隣接行列は A_3 と A'_3 の 2 種類ある．

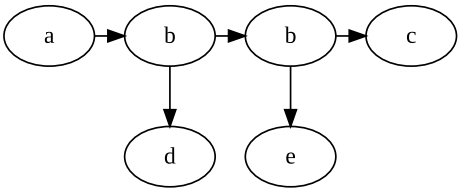

図 2 (2) の例
Fig. 2 Example of (2)

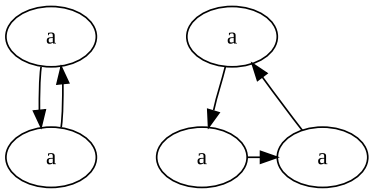

図 3 (3) の例
Fig. 3 Example of (3)

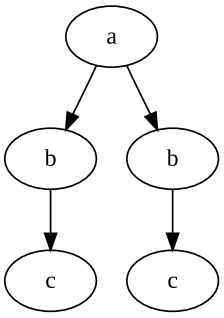

図 4 (4) の例
Fig. 4 Example of (4)

$$A_3 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad A'_3 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

2.3 提案手法

本研究では随時投入されるアプリから定義ファイルを自動的に作成して疑わしいアプリを絞り込む方法を提案する．アプリは 2.1 節で説明したメソッドの制御フローのグラフで構成される．メソッドを m とすると，アプリ $X = \{m_{X1}, m_{X2}, \ldots, m_{Xn}\}$ と定義ファイル $D = \{m_{D1}, m_{D2}, \ldots, m_{Dn}\}$ は 1 つ以上のメソッドから構成される．またマルウェアではないアプリに含まれているメソッドの集合 $B = \{m_{B1}, m_{B2}, \ldots, m_{Bn}\}$ を投入されたアプリから構築する．

2.3.1 包含度

アプリ X のメソッドに定義ファイル D のメソッドが含まれる割合を求め，その値を包含度とする．メソッドに含まれるノードの数を $|m|$ とすると，包含度は

$$I = \frac{\sum_{m \in D \cap X} |m|}{\sum_{m \in D} |m|} \times 100$$

と定義される．アプリとすべての定義ファイルの包含度を求め，最も大きい包含度をそのアプリの包含度とする．

2.3.2 定義ファイル作成

アプリがマルウェアである場合には，マルウェアだけに

含まれているメソッドから定義ファイル D を作成する．新たに作成する定義ファイル D はアプリ X から B を除いた差集合 $D = X - B$ となる．

新たに作成した定義ファイル D に対してすべての既存の定義ファイル D_i と重複するメソッドの量

$$S = \max_{1 \leq i \leq n} \sum_{m \in D \cap D_i} |m|$$

を求める．S が 0 でないならば，S が最大になる既存の定義ファイル D_i との積集合 $D \cap Di$ を新たな D_i とし，新たに作成した定義ファイル D を破棄する．S が 0 ならば新たに作成された定義ファイルは保存される．

2.3.3 定義ファイル再構築

アプリがマルウェアではない場合には，B にアプリ X のメソッドを加える．またすべての定義ファイル D からアプリ X のメソッドを取り除く．このとき，定義ファイル D が空集合になったならば，その定義ファイル D の元になったすべてのアプリに対して定義ファイルの作成をやり直す．

2.3.4 閾値

包含度に対して閾値を設定し，包含度が閾値よりも大きいときにはマルウェアとみなし，閾値以下のときには通常のアプリとみなす．それまでの結果から最も誤りが少なくなる値を閾値とする．閾値に近い包含度をもつアプリは誤って判別されている可能性が高いと考え，閾値との差により解析すべきアプリの順番を決定する．閾値には偏りがあるので，閾値と包含度の最大値 100 および最小値 0 との差に対する比率で閾値との近さを計る．たとえば閾値が 20 ならば，包含度が 10 のアプリと 60 のアプリは閾値からの近さは同じになる．

3. 実験

本研究では既に解析済みの検体セットから定義ファイルを自動的に作成する．定義ファイルを作成する過程において，アプリの包含度を随時算出する．最後に算出した包含度から閾値を設定し，仮にその閾値で判別したときに，どの程度のアプリが正しく判別できるのかを検証する．また誤って判別したアプリを訂正するためには，どの程度の解析が必要になるのかを見積もる．

3.1 マルウェアの定義

本研究では次の条件のいずれかを満たす Android のアプリをマルウェアとする[*3]．

- □ ユーザの意図に反する動作を故意に行うアプリ
- □ 合理的な理由なく高額な課金をされるアプリ
- □ 脆弱性を利用するアプリ
- □ 専ら非合法な目的で利用されると考えられるアプリ
- □ 他のマルウェアと密接に関連するアプリ

[*3] 参考文献 [2] とはマルウェアの定義が異なる

3.2 検体セット

拡張子が.apk であり ZIP のアーカイブ内に classes.dex があるという条件で，我々が 2011 年 6 月から 2013 年 9 月までに Web サイトから収集したファイルは 151,952 種類あった．これらのうち署名を検証でき，制御フロー解析を行うことができ classes.dex が重複しないアプリは 126,990 種類あった．126,990 種類のアプリのうち 1,522 種類がマルウェアであることがわかっている．

アプリから抽出されるグラフは隣接行列で表す．また文字列の比較には時間がかかるので，グラフのノードのラベルは 32 ビットのハッシュ値で扱い，ラベルはハッシュ値の小さい順に並べる．ゆえに，隣接行列とハッシュ値の数列をあわせたバイト列がメソッドから抽出された特徴になる．1 つのメソッドのバイト数はノードの数を n とすると次の式で求めることができる．

$$len = \lceil \frac{n^2}{8} \rceil + n \times 4$$

3.3 グラフ比較の検証

2.2 節で述べたように，隣接行列が一意に決定できない場合がある．そこで 126,990 種類のアプリの中から 10,000 種類のアプリをランダムに選び，グラフを一意に決定できないものが，どの程度存在するのか検証を行った。同じアプリに同じグラフが含まれているときにには 1 つにまとめ，異なるアプリにある同じグラフはそれぞれを 1 つのグラフとして数えたところ，抽出したグラフは全部で 30,374,116 あった．30,374,116 の中で隣接行列が一意に決定できないグラフは延べ 91 であった．よって，本研究では隣接行列が一意に決定できないグラフの数は無視できる程に僅かであると考え，本研究ではバイト列を比較することでグラフの比較を行う．

3.4 包含度算出

表 2 はアプリ内部の classes.dex のファイルの日時をアプリの作成日時とみなし，アプリを古い順から本研究が提案する手法で定義ファイルを作成し，包含度を算出した結果である．表 2 は包含度の値の範囲と，その範囲にあるマルウェアと通常のアプリの数を表している．

図 5 は包含度が 100 未満のマルウェアを赤の点で，包含度が 0 より大きい通常のアプリを青の点で表している．包含度が 100 のマルウェアおよび包含度が 0 の通常のアプリの包含度は適切であり，また大量にあるので図 5 に描画されていない．包含度が大きいアプリほど図 5 の上に配置され，最も上の点は包含度が 100，最も下は 0 になる．図 5 では包含度を算出した順番で左から順に点は配置されており，日時が最も古い描画の対象となる検体の点が最も左に，

表 2　包含度とアプリ数
Table 2　Inclusion Degree and the Number of Apps

	Malware	Benign
$I = 100$	1,035	23
$0 < I < 100$	236	813
$I = 0$	251	125,468

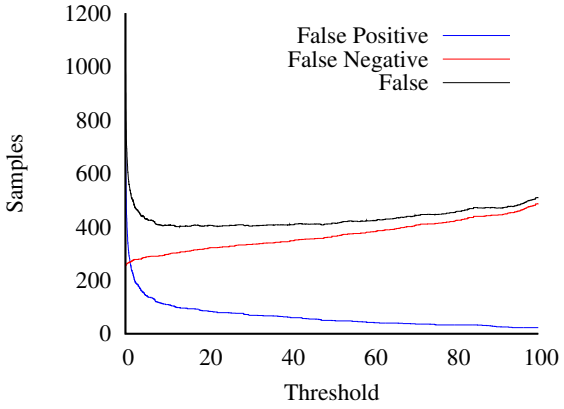

図 6　閾値と誤り数
Fig. 6　Threshold and the Number of False

表 3　閾値と最小の誤り数
Table 3　Threshold and the Minimum Number of False

Threshold	False Positive	False Negative
$12.19 \leq T < 12.49$	99	302
$12.49 \leq T < 13.15$	98	303
$13.19 \leq T < 13.2$	97	304

最も新しいアプリの点が最も右にある．

3.5　閾値設定

閾値に対する False Positive（偽陽性）と False Negative（偽陰性），False（偽陽性と偽陰性の合計）は**図 6** になる．誤って判別されたアプリの数の最小値は 401 で，このときの閾値の範囲と誤りの数を**表 3** に示す．最小となる閾値の範囲の中間の値である 12.67 を閾値としたとき，閾値に近い包含度をもつアプリから順番に解析者が解析を行い誤りを訂正した場合，解析するアプリの数と訂正できる誤りの数を**図 7** に示す．

3.6　定義ファイル作成

実験の結果，249 種類の定義ファイルが作成された．これらの定義ファイルに対応する検体の数と定義ファイルに含まれるメソッドの数を**図 8** に示す．1 つの検体で 1 つメソッドから作られた定義ファイルが 21 種類で最も多かった．また対応する検体が 1 つしかない定義ファイルは 59 種類であった．最も対応する検体数が多い定義ファイルは 278 検体に対応しており，最も多くのメソッドを含む定義ファイルは 1,243 のメソッドを含んでいた．

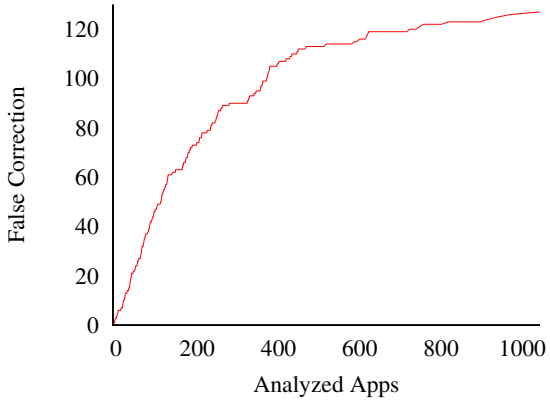

図 7　解析アプリ数と誤り訂正
Fig. 7　Analyzed Apps and False Correction

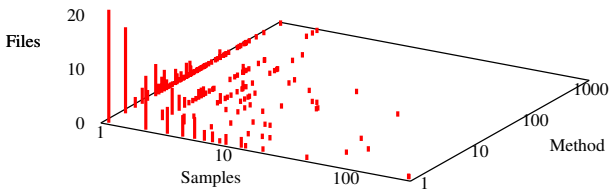

図 8　定義ファイル
Fig. 8　Definition File

マルウェアのうち 22 種類は，検体の中のすべてのメソッドが通常のアプリでも使われているため定義ファイルの作成ができなかった．22 種類のうち 1 種類は一度は定義ファイルが作られ，別の 1 種類は既存の定義ファイルで包含度が 100 になったが，2.3.3 節の再構築で定義ファイルが空集合となり，その後に定義ファイルを再度作成することができなかった検体である．

4.　考察

自動的に作成された定義ファイルにより，1,522 種類のマルウェアのうち約 $\frac{2}{3}$ に相当する 1,035 種類を検出できることがわかった．また閾値を設けた場合には，誤って判別された数は 401 種類であった．アプリの包含度と閾値の近さから解析すべきアプリの順位を決めた場合，包含度が $0 < I < 100$ であるアプリの約 $\frac{1}{5}$ を解析することで約半分の誤りを訂正できる．しかし 401 種類中 251 種類は過去のマルウェアと共通点がなかったので包含度が 0 となり，提案する方法では見つけることはできない．新種を発見するためには他の方法と組み合わせる必要がある．

401 種類中 23 種類は包含度が 100 になったものの通常のアプリであった．また 22 種類はマルウェアのメソッドのすべてが通常のアプリでも使われていたため定義ファイルを作成できなかった．これらのアプリについても他の方法と組み合わせる必要がある．

3.1 節でマルウェアの定義を示しているが，本研究で提案する手法には本研究のマルウェアの定義に基づいてマル

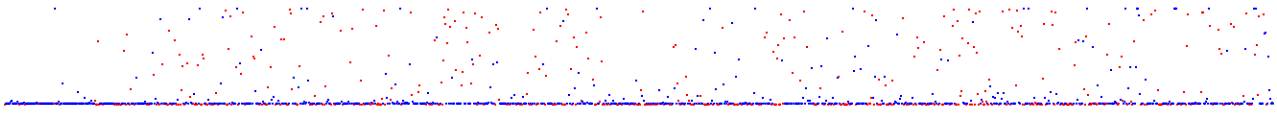

図 5　包含度の分布
Fig. 5　Distribution of Inclusion Degree

ウェアを見つける仕組みは備わっていない．提案する手法は既に解っている過去の結果からアプリを判別しているので，3.1 節のマルウェアの定義以外の基準を用いることもできると思われる．

定義ファイルを解析者が作成する場合には，マルウェアの定義に関係するメソッドを定義ファイルに含める．一方，本研究ではマルウェアだけで使われているメソッドのうち共通するメソッドを定義ファイルに含めているが，定義ファイルに含まれているメソッドが 3.1 節で示すマルウェアの定義に関わるメソッドであるか否かは検証されていない．しかし本研究の目的は 3.1 節のマルウェアの定義に基づいて定義ファイルを自動的に作成することではないので，定義ファイルの構成は問題ではない．

本研究の実験では最終結果から閾値を決め，過去のアプリにもその閾値を用いている．しかし現実の運用ではその都度，閾値を求め直す必要がある．

本研究の実験では過去に解析済みのアプリを用いており，包含度による判別に誤りがあったときには直ちに訂正している．しかし現実の運用では解析あるいは他の方法による検知などが必要であり，直ちに訂正することはできない．

5. 関連研究

Burguera らはクラウドを利用したマルウェアを検出する方法 [3] を提案している．また Shabtai らは機械学習によりアプリの動作からマルウェアを検出する方法 [4] を提案している．我々の研究とは異なりこれらの方法は動的解析であるため，マルウェアが実行されなければ検出できない．一方，バイトコードが異なる場合でもマルウェアとしての動作があるならば，未知のマルウェアを検出できる可能性がある．

Thomas らは動的解析であるサンドボックスに静的解析を組み合わせる方法 [5] を提案している．この研究ではサンドボックスで検出できない脅威を，逆アセンブル結果からいくつかのパターンを探すことで検出することを試みている．

Pouik らは Android のアプリにおいて，Normalized Compression Distance（NCD）を用いて類似度を算出する方法 [6] を提案している．この方法も我々の提案と同じく静的解析を行っているが，類似度の算出を NCD で行う点は我々の提案とは異なる．

本研究の提案は我々が過去に提案した方法 [2] ではマルウェアの検出を目的としているのに対して，本研究では解析者が解析するべきアプリを絞り込むことを目的としている．本研究の提案では参考文献 [2] の検討課題にあった定義ファイルの作成の自動化を行っている．参考文献 [2] と包含度や閾値などの基本的な考え方は同じであるが，参考文献 [2] は 2011 年の我々の提案 [7] 以降，随時解析を行って定義ファイルを作成し，その時の状況に応じて閾値などを導入してきた．それに対して本研究ではこれまでの結果を再度検討し直して，マルウェアの定義を明確にし，閾値は定義ファイル毎に設定せずに全体に対して 1 つの値を設定するようにしている．

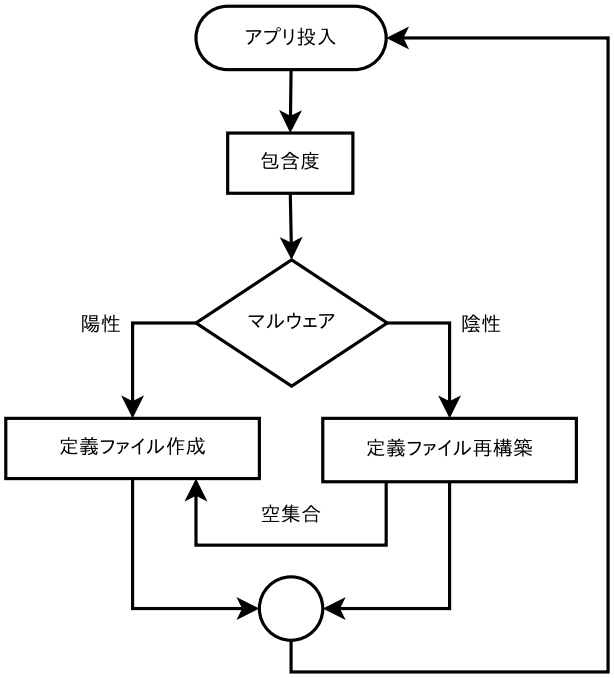

図 9　フローチャート
Fig. 9　Flowchart

また制御フロー解析を行い結果のグラフを特徴とするという考え方は，我々が Windows においてマルウェアを分類したときの方法 [8] を Android に応用したものであるといえる．

6. 今後の課題

本研究では解析すべき疑わしいアプリを絞り込むことを目的とした．今回は既に解析済みの検体セットを用いており，解析の結果から定義ファイルの作成を行うか再構築をするのかを決定していた．しかし実際の運用では，**図 9** のように包含度と閾値による判別の結果から定義ファイルの作成または再構築を行う．

今後は本研究で扱った正常なアプリのメソッドおよびマルウェアのメソッドから抽出した特徴を応用して，解析すべきアプリの中でどのメソッドを見るべきかを案内できるようなシステムも検討したい．Android のアプリには，広告のモジュールなど多くのアプリで共通して使われるバイトコードが含まれていることが多く，そのような解析する必要がないバイトコードとアプリ固有のバイトコードを分離することができれば，解析の効率が良くなることが期待できる．

またマルウェア固有のメソッドから抽出した特徴を利用して，マルウェアを分類することも検討したい．未知のマルウェアの検体が，すでに解析されているどのマルウェアに似ているかを知ることができれば，検体の動作や構造を推定できるため, マルウェアの解析を行ううえで手がかりとなる．

本研究で提案する手法では新種を発見できないことは明らかなので，他の方法と組み合わせる必要がある．5 節で挙げたマルウェアの動作に基づいて検出するような方法や，あるいは我々が検証したコード解析に依らない方法 [9] が考えられる．

参考文献

[1] Dirro, T., Greve, P., Li, H., Paget, F., Pogulievsky, V., Schmugar, C., Shah, J., Sherstobitoff, R., Sommer, D., Sun, B., Wosotowsky, A. and Xu, C.: McAfee Threats Report: Second Quarter 2013, McAfee Labs (online), available from ⟨http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2013.pdf⟩ (accessed 2013-11-12).

[2] 岩本一樹，和崎克己：制御フロー解析により生成されたグラフ比較による Android マルウェア検出方法の提案，研究報告コンピュータセキュリティ（CSEC） 5 (2013).

[3] Burguera, I., Zurutuza, U. and Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for Android, *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, New York, NY, USA, ACM, pp. 15–26 (online), DOI: 10.1145/2046614.2046619 (2011).

[4] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C. and Weiss, Y.: "Andromaly": a behavioral malware detection framework for android devices, *Journal of Intelligent Information Systems*, Vol. 38, No. 1, pp. 161–190（オンライン），入手先 ⟨http://www.springerlink.com/index/10.1007/s10844-010-0148-x⟩ (2012).

[5] Bläsing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A. and Albayrak, S.: An Android Application Sandbox system for suspicious software detection, *In Proceedings of the 5th International Conference on Malicious and Unwanted Software (Malware 2010)* (2010).

[6] Pouik and G0rfi3ld: Similarities for Fun & Profit, Phrack Inc. (online), available from ⟨http://www.phrack.org/issues.html?issue=68&id=15#article⟩ (accessed 2013-11-12).

[7] 岩本一樹，和崎克己：制御フロー解析による Android マルウェア検出方法の提案，コンピュータセキュリティシンポジウム 2011 論文集，Vol. 2011, No. 3, pp. 714–719 (2011).

[8] 岩本一樹，和崎克己：静的解析により抽出された API 推移に基づくマルウェアの分類，情報処理学会論文誌， Vol. 54, No. 3, pp. 1199–1210 (2013).

[9] 岩本一樹，西田雅太，和崎克己：コード解析を伴わない Android マルウェア検出方法の検証，研究報告コンピュータセキュリティ（CSEC） 60 (2013).