Android マルウェアの API 処理ロギングによる個人情報漏えい検知及び分類

韓, 燦洙
九州大学

松本, 晋一
九州先端科学技術研究所

川本, 淳平
九州大学

櫻井, 幸一
九州大学大学院システム情報科学研究院 : 教授

出版情報：火の国情報シンポジウム. 2016, pp.3A-4-, 2016-03-02. 情報処理学会九州支部
バージョン：
権利関係：

Androidマルウェアの API呼出し記録による分類及び個人情報漏えい検知

韓 燦洙[1,a)]　松本 晋一[2,b)]　川本 淳平[1,2,c)]　櫻井 幸一[1,2,d)]

概要：本稿では，API 呼出し記録による Android マルウェア分類手法を提案する．Android マルウェアの動的解析としてインスタンスからシステムコールを取得する．そして，各システムコールを特徴とし，呼び出された回数を値として特徴ベクトルを作成する動的解析を行う既存研究の手法がある．しかし，システムコールでは特定できない，プロセス間通信だけで機微な情報を取得する API が存在することから，API 呼び出しについても特徴として考慮する必要があると考えられる．また，API は情報を豊富に含んでいるので，本研究ではシステムコールと API を特徴として組合せる．従って，より良い精度でマルウェアを解析・分類する新たな手法を提案し，実験評価した．

Classified by the API call record and personal information leakage detection of Android malware

Chansu Han[1,a)]　Shinichi Matsumoto[2,b)]　Junpei Kawamoto[1,2,c)]　Kouichi Sakurai[1,2,d)]

Abstract: In this paper, we propose a Android malware classification technique by the API call record. There is a technique of prior research that get the system call from the instance as dynamic analysis of Android malware. Replace each system call as feature, and create a feature vector from the called number of times. However, there is the API which call the sensitive information in the only communication between processes which can't identify API by system calls. Since, it is considered that it is also necessary to take an API call as a feature. In addition, the API contains the information rich, in this study by combining the system calls and API as the feature value. We propose a new method to analyze and classify malware in better accuracy, and experimental evaluation.

1. はじめに

世界シェア 1 位のスマートフォンプラットフォームである Android は，使用率が毎年増加している．Android プラットフォームでは，オープンソースであるため誰でも容易にアプリケーションを開発し公開できる．また，ユーザは自由にアプリケーションをダウンロードし，利用できる．しかし，市場シェアが高いこと，オープンソースであること，非公式マーケットでアプリケーションがダウンロードできることから Android を対象としてモバイルマルウェアが数多く作成されている．実際，現在毎日平均として 6,000 個以上の新しい Android マルウェアインスタンスが検出されている [3]．マルウェアは主に非公式マーケットから拡散していくが，公式マーケットでもマルウェアが検出されている [4]．そのため，ユーザと開発者以外の第 3 者である公式マーケット運営者によるマルウェア検知も重要になってきている．

マルウェアの解析技術には静的解析と動的解析の二つがある．静的解析では，アプリケーションを逆コンパイルしソースコードを調べる．そして，パターンマッチングによりマルウェアを検出する．この方法では，実際にアプリ

[1] 九州大学
Kyushu University
[2] 九州先端科学技術研究所
Institute of Systems, Information Technologies and Nanotechnologies(ISIT)
[a)] 1TE12190K@s.kyushu-u.ac.jp
[b)] smatsumoto@isit.or.jp
[c)] kawamoto@inf.kyushu-u.ac.jp
[d)] sakurai@csce.kyushu-u.ac.jp

ケーションを動かすことなく解析が可能である．マルウェアの被害にあう前に潜在的な脅威を検知できる．しかし，アプリケーションのソースコードが難読化されている場合やアプリケーションが外部サーバと連携を行い，攻撃コードを実行時にダウンロードする場合，またはマルウェアの新種・亜種に対するパターン生成が追い付かないほど短期間に大量に発生した場合などはマルウェアを見逃してしまう可能性が高くなる．一方，アプリケーションを実行して挙動を調べる動的解析は静的解析の問題を解決できる．動的手法は静的解析と異なりアプリケーションを実行する必要はあるが，実際の挙動を調べることでマルウェアか否かを判断する手法である．

本研究では日々 6,000 個以上のマルウェアアプリケーションが発見されている現在では新種・亜種に対する解析は静的解析では追い付いていけないと考えられる．また，マーケットの運営者側が事前にマルウェアを検知する事よりユーザはマルウェアの被害にあわなくなるため，静的解析を用いることなく動的解析を用いた解析手法に着目した．

Burguera らは，Android においてシステムコールを動的解析するマルウェア検知手法を提案している [1]．彼らの手法は，Linux のデバッグツールである Strace を用いてシステムコールを監視している．そして各システムコールを特徴とし，呼び出された回数を値とする特徴ベクトルを作成する．この特徴ベクトルに K-means クラスタリングアルゴリズムを適用し，非正常な動きをするアプリケーションを分類している．

一方，西本らは Android にはカーネルサービスを利用せずにプロセス間通信より機微な情報を呼び出す API（Application Programming Interface）が含まれていることを確認している [2]．つまり，カーネルサービスの利用情報であるシステムコールを使わず悪意ある行動を起こすアプリケーションが存在することを意味する．しかし，本実験で実際に API を呼び出すアプリケーションを作成して確かめてみたところ，実験環境に差があったためなのか，ioctl と mprotect システムコールを呼び出していた．ioctl はデバイスドライバと通常のデータの読み書きの流れの外で通信するために用意されたシステムコールであり，mprotect はメモリ領域のアクセス許可を制御するシステムコールである．これらは Android ではプロセス間通信を行うには Binder ドライバを利用するので，Binder ドライバを利用する際に呼び出されるシステムコールだと考えられる．従って，API を呼び出すには Binder ドライバを利用するため，2 種類のシステムコールを呼び出していることが確認できた．しかし，API 呼び出しだけがプロセス間通信を利用することではないので，Binder ドライバを利用したことだけでは API の特定は難しい．API の特定が難しいことは API に含まれている豊富な情報の中でどんな情報を取得されたかわからない．本実験ではそういった API を実験対象とする．

Burguera らの手法では，システムコール呼び出し回数のみを特徴として解析するため，API を用いた情報呼び出しに対しては検知できていないと考えられる．例えば，連絡先を悪意のある意図で呼び出し，外部に送信するマルウェアを考える．マルウェアが取得した情報を外部に送信するときやファイルに書き込むときはシステムコールを利用する．システムコールだけでは連絡先の情報を管理しているプロセスからアプリケーションのプロセス間の通信より呼び出した情報が連絡先の情報なのかは判定できない．従って，システムコール情報のみから正規のアプリケーションだと分類されても，実は情報漏えいしている可能性がある．言い換えれば，システムコール呼び出し回数のみを特徴として用いる動的解析は不十分である．実際に，本研究で行った実験では，結果としてシステムコール呼び出し回数のみを特徴とする場合より API 呼び出し回数も特徴に含めた方が良い分類を見せている．また，システムコール呼び出し回数のみからは，どのような脅威が潜在しているか具体的にわからない．しかし，それぞれの API はどのような情報を取得するのか公開されているため，API の呼び出しを見ることより具体的にどのような情報が脅威にあったか確認できる．

本論文では，新たにシステムコールでは特定できない API の情報も特徴として用いる動的解析手法を提案する．API 呼出しを記録するには，Android OS のフレームワークを改変，具体的には Log メソッドを挿入して再ビルドする必要がある．Android をカスタマイズする必要があるが，アプリケーションマーケット運営者側が登録要請されたアプリケーションを検査する場合に利用すると考えると問題にならないと考えられる．日本の場合は各キャリアごとにアプリケーションマーケットがあるので，こういったマーケットの運営者側が活用することを想定している．また，流通前のアプリケーションは新種や亜種のマルウェアが混入していることが多い．新種や亜種のマルウェアが日々作成される状況では，教師データの作成が難しく教師あり学習は適さない．従って，本実験では大量の新種・亜種に対応するために教師有り学習ではなく教師なし学習の K-means クラスタリングアルゴリズムを用いることにする．以上より，システムコールと API を特徴として組合せることで，より良い精度でマルウェアを解析・分類する新たな手法を提案する．

本論文の構成は第 2 節では，Android におけるマルウェアの解析に関する関連研究と既存研究について記す．第 3 節では，既存研究が持つ不十分な手法に対する提案手法を記す．第 4 節では，データ収集のための具体的な実験手順を記す．第 5 節では，マルウェア分類と情報漏えい検知の実験結果，評価及び考察を記す．第 6 節ではおわりに実験のまとめと今後の課題をまとめる．

2. 関連研究

Android におけるマルウェアの解析・分類には様々な検出手法が研究されている．Faruki ら [7] は最新の Android のセキュリティについて議論し，既存の検出首の長所と短所などをまとめている．磯原ら [5] によると Android マルウェアは大きく攻撃型，特権利用型，情報漏えい型 3 種類に分類される．正規表現を用いて「攻撃ツールの名称」，「通信の宛先情報」，「特権を必要とするコマンド」シグネチャより攻撃型と特権利用型マルウェアを検知する手法を提案している．磯原らの論文では情報漏えい型マルウェアの検知には限界があるため，本研究では情報漏えいの検知手法に着目した．

アプリケーションをリバースエンジニアリングすると，Android マニフェストファイルがあり，その中にはパーミッションなどの細部な情報が入っている．パーミッションから呼び出される API を抽出し，特徴とした解析 [14], [16], [19], [22] に関する研究が多数存在する．リバースエンジニアリングより，クラス名，メソッド名などから API 呼び出しリスト作成し，機微な情報を呼び出すなどの API に高いレベルを付けインスタンスを学習させる研究が多い．また，パーミッション，API，システムコールなど色んな情報を組合せてから確率論に基づくアプローチより検出するものもある [13].

API フックによる API 呼び出しを取得し，マルウェアと正規のアプリケーションの挙動から API 呼び出しの違いを分析，教師有り学習する研究が主流になっている [8], [15]. 笠間ら [6] は同じインスタンスを複数回実行し，毎回 API のログを取って差異を調査する手法を提案している．しかし，API フックではアプリケーションのプログラムで API を利用せずにその API と同じ働きをするメソッドを定義する場合，API 呼び出しは取得されないと考えられる．API 呼び出しを確実に記録するには，アプリケーションのプロセスと同プロセスで実行されるクラスのメソッドではないことが必要である．API など機微な情報を呼び出すとき libc 関数が横取りすることから組合せた学習手法 [9] はその対案となるが，本研究では次の西本論文 [2] を参考に進めている．

2.1 端末情報の取得検知手法

西本ら [2] は，システムコールを利用することなくプロセス間通信だけで端末情報を取得する API の存在を検知・確認している．また，実際に API を用いて端末情報を取得するアプリケーションを作成し，そのアプリケーションから端末情報を取得する API を実行する際のシステムコールのログを Strace を利用して確認・検証している．実験対象としては TelephonyManager クラスにあるいくつかの API に対して実験を行った結果，API を呼び出す際にシステムコールを利用しない API をいくつか特定できた．端末識別番号を取得する getDeviceId API におけるプロセス間通信の概略図を**図 1** に示す．Binder ドライバのクラスよりプロセス間通信を行い，端末識別番号をアプリケーションが取得することができる．しかし，本研究での実験環境では API を呼び出す際にシステムコールを利用していることが確認できた．詳細は 4 節に記す．

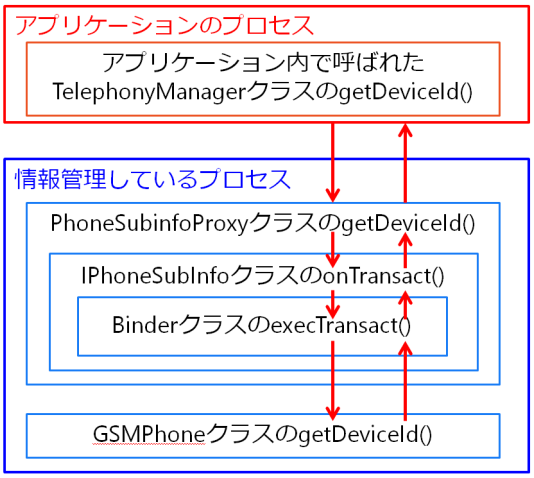

図 1 getDeviceId() におけるプロセス間通信の概略図
Fig. 1 Schematic of inter-process communication in getDeviceId()

2.2 クラウドソーシングを用いた挙動ベースマルウェア検知手法

Burguera ら [1] は，strace を用いてアプリケーションを実行したときに使用されたシステムコールを取得し，その情報から動的解析するマルウェア検知手法を提案している．彼らの提案では，各システムコールを特徴に，呼び出された回数を特徴量とする特徴ベクトルを作成する．この特徴ベクトルから k-means クラスタリングアルゴリズムを適用し，マルウェアと正規のアプリケーション二つのクラスターに分類している．以上より，非正常な挙動をするアプリケーションを検知している．また，データを収集するためにクラウドソーシングを用いている．クラウドソーシングとは不特定多数の第三者である人間にある作業を委託するという雇用形態を指す．Crowdroid というアプリケーションを開発，公開し多くのユーザからアプリケーションのシステムコールログを中央サーバに収集し，提案している手法より解析することである．この手法の特徴として，Crowdroid のユーザが多ければ多いほどより正確なデータセットが作成されることがあげられる．実験結果としては，研究者本人が作成したマルウェアアプリケーションサンプルに対しては 100%の検知率と PJApps マルウェアに対しても 100%の検知率，トロイの木馬型の HongTouTou マルウェアに対しては 85%という検知率を得ている．その他にも，機微なシステムコールを選定し，そのシステム

コールのみを特徴とした教師有り学習による自動分類手法もある [10].

3. 提案手法

Burguera らの手法では Strace で取れるシステムコール呼び出し回数のみを特徴として分類・検知を行うが，実際にはプロセス間だけで通信を行う API の存在がいくつか確認されている．API の呼び出された情報に対してはシステムコールだけでは特定できないため，Burguera らの手法では特定の API 呼び出し検知はできないと考えられる．従って，システムコール呼び出し回数のみを特徴とする手法で正規アプリケーションだと分類したインスタンスが実は API を悪用していて情報漏えいしている場合も考えられる．

本実験では二つの既存研究を参考にして，新たなマルウェアの解析・分類する手法を提案する．まず，API 呼び出しを記録する環境にするために Android OS の改変を行う．Android エミュレータ上からアプリケーションインスタンスを実行し，Strace と Logcat を用いてシステムコールごとの呼び出された回数と API ごとの呼び出された回数を記録する．そしてその呼び出された回数を特徴量とし，特徴ベクトルを作成する．インスタンス一つに対して特徴ベクトルを一つ作成する．その特徴ベクトルの次元は Linux カーネルバージョン 3.4.0 のシステムコール数が 394 個，実験対象の API が 8 個で計 402 次元となる．Linux バージョンごとのシステムコールの数は Linux の公式サイトなどで確認できる [12]．次に特徴ベクトルの例を記す．特徴の順番はシステムコールのアルファベット順の次に API の code 番号順に並べた．例から見るとシステムコールの access が 15 回，brk が 89 回，cacheflush が 837 回，chmod が 5 回のように呼び出された回数を値に特徴ベクトルを作成する．

```
0,0,0,0,0,15,0,0,0,0,0,0,0,0,89,837,0,0,0,5,0,0,0,0,0,32436,
0,0,72,188,0,0,0,0,73,0,0,0,0,26,0,6246,…
```

マルウェアか否か検査すべきアプリケーションを複数個集めて挙動より特徴ベクトルを作成する．それから，K-means クラスタリングアルゴリズムを用いてマルウェアを分類をする．二値分類のため K=2 に設定する．既存研究の Crowdroid との相違点は，Crowdroid は Strace から取得するシステムコール呼び出し回数のみを特徴とするが，今回の実験ではシステムコールでは特定できない API のことに着目し，その API も特徴とする．

4. 実験手順

API 呼出し記録

求める API のログを取得するためには AndroidOS のフレームワークを改変する必要がある．IPhoneSubInfo.java クラスの onTransact メソッドの内部にログを出力するように Log.v を挿入する．Log.v を挿入する場所を決めた条件としては，アプリケーションのプロセスと同プロセスで実行されるメソッドでないこと，メソッドを呼び出した API が特定できること，そしてできるだけ多くの API の呼出しが検知できるメソッドであることから IPhoneSubInfo.java クラスの onTransact メソッドが適当であると述べられている [2].

```
access("/data/data/com.example.testbutton/files", F¥_OK) = 0
open("/data/data/com.example.testbutton/files/myfilelog.txt",
write(42, "getDeviceId¥_begin¥|¥|n", 18) = 18
close(42) = 0
ioctl(9, 0xc0186201, 0xbeb943e8) = 0
...ioctl 14回
mprotect(0xb2d97000, 4096, PROT¥_READ|PROT¥_WRITE) = 0
ioctl(9, 0xc0186201, 0xbeb943e8) = 0
...ioctl 8回
access("/data/data/com.example.testbutton/files", F¥_OK) = 0
open("/data/data/com.example.testbutton/files/myfilelog.txt",
write(42, "getDeviceId¥_done¥|¥|n", 17) = 17
close(42) = 0
```

図 2 Strace を用いてシステムコールのログ例
Fig. 2 Example of system call logs by using Strace

Android 改変手順

Android OS をビルドするには Linux OS 上でビルドする必要がある．本実験では Oracle VirtualBox 上の仮想マシンとして動作する Ubuntu 14.04 を用いた．また，AndroidOS はバージョンによって API Level が異なる．今回は，2015 年 12 月時点で最も利用率の高かった API Level 19 であるバージョン 4.4 KitKat を選んだ [17]．以下に Android OS の改変手順を示す．Android Open Source Project の手順通りに Android OS ソースをダウンロードする [11]．入手したソースコードを make コマンドでビルドする．ビルドをすると IPhoneSubInfo.aidl ファイルから自動的に IPhoneSubInfo.java ファイルが作成される．作成された IPhoneSubInfo.java の onTransact メソッド内に Log.v を挿入する．make コマンドで再ビルドする．エミュレータ上で IPhoneSubInfo 内の API を呼び出すアプリケーションを実行する．Logcat を利用して API のログを確認する．並びに，Strace を用いてそういった API 呼び出しに呼び出されるシステムコールを確認できる．

今回の実験ではテスト用としてボタンを押すと IPhoneSubInfo 内の API を順番に呼び出すような簡単なアプリケーションサンプルを作成した．そして，作成した APK ファイルをエミュレータにインストールし，実行した．API を呼び出すボタンを押す前に Strace でアプリケーションのプロセスにアタッチし，システムコールをファイル出力する．また，Logcat を利用して API 呼出し記録ができたか否かを確認する．次の**図 2** に実際にボタンを一回押したときのシステムコール例を示す．ボタンを押すと API を呼び出す前後にファイルに何か書き込むプログラムである．図 2 の囲まれた部分がファイル操作の部分で，間のシステムコールが API を呼び出した際に呼ばれたものである．

表 1　IPhoneSubInfo クラスでの code 変数の対応表（API Level 19）

Table 1　code variable correpondence table of IPhoneSubInfo class.(API Level 19)

code	API	○/×
1	getDeviceId()	○
2	getDeviceSvn()	○
3	getSubscriberId()	○
4	getGroupIdLevel1()	○
5	getIccSerialNumber()	○
6	getLine1Number()	○
7	getLine1AlphaTag()	×
8	getMsisdn()	×
9	getVoiceMailNumber()	○
10	getCompleteVoiceMailNumber()	×
11	getVoiceMailAlphaTag()	○
12	getIsimImpi()	×
13	getIsimDomain()	×
14	getIsimImpu()	×

API を呼び出す際に ioctl が 24 回と mprotect が 1 回呼び出されている．しかし，これらは Android でプロセス間通信を可能にする Binder ドライバが呼び出したシステムコールである．ioctl はデバイスドライバと通常のデータの読み書きの流れの外で通信するために用意されたシステムコールで，図 1 の Binder ドライバクラスがプロセス間通信をするために ioctl を呼び出したと考えられる．mprotect はメモリ領域のアクセス許可を制御するシステムコールで，つまり Binder ドライバはカーネルサービスを利用していると考えられるが，API そのものはカーネルサービスを利用していないことがわかる．従って，このような API はシステムコールでは特定できない．

また，**表 1** に API Level 19 での IPhoneSubInfo クラスでの code 変数の対応表を示す．○がついてる API は TelephonyManager クラスに属する API で今回の実験対象となる 8 個の API である．×がついてる API は IPhoneSubInfo クラスには定義されてはあるが，TelephonyManager クラスに属していないため実験対象外とする．

データ収集・マルウェア分類

本節ではどのようなインスタンスを集め，データを収集し，解析したかについて述べる．Google 社の公式マーケットである Google Play から 9 個のアプリケーションをダウンロードし，マルウェアデータセットを集めているサイト [20] から 15 個のアプリケーションをダウンロードして実験対象とした．計 24 個のインスタンスを評価クラスを付けるために VirusTotal スキャン [21] を用いて事前に Google Play からダウンロードした 9 個のアプリケーションは正規のアプリケーションであることと，マルウェアサンプルは実際に悪性アプリケーションであることを確認した．

本実験ではすべてのデータ収集をエミュレータ上で行う．Linux 3.4.0 のシステムコールの数は 394 個であり，表 1 のなかで TelephonyManager クラスに属する 8 個の API をもとに特徴ベクトルを作成する．計 402 次元で各システムコールや API が呼び出された回数を特徴量とする．インスタンスごとに実行開始から約 5 分間 Strace と Logcat を用いてシステムコールと API の呼び出し回数を収集した．Strace を用いて収集したシステムコールのデータにはエラー回数の情報も含まれている．そこで，システムコールの呼び出された回数のみを特徴量とする場合とシステムコールの呼び出された回数とエラーの呼び出された回数を特徴量とする場合，システムコールの呼び出された回数にエラーの呼び出された回数を引いたものを特徴量とする場合の 3 つの CASE に分けてそれぞれ比較を行った．解析には K-means クラスタリングアルゴリズムを適用し，正規アプリケーションかマルウェアかの 2 つのクラスターに分ける分類を行った．また，それぞれの CASE のデータから API の特徴がない場合とある場合，そして API 呼び出し回数のみの場合を実験評価し，主成分分析を行い次元圧縮する場合としない場合を実験評価した．

5. 実験結果，評価及び考察

5.1 マルウェア分類

収集したデータからシステムコールのエラーを様々な方法で取り込み K-means クラスタリングアルゴリズムを適用した結果を**表 2** に示す．CASE の定義は下記する．CASE それぞれのデータに API の呼び出された回数を特徴量として入れた場合と入れてない場合に分け，さらに主成分分析 (PCA, Principal Component Analysis) を用いて次元圧縮する場合と次元圧縮を行わない場合に分けてクラスタリングを行った結果である．

CASE の定義

CASE1
システムコールの呼出された回数のみを特徴量とする場合

CASE2
システムコールの呼出された回数とエラーの呼び出された回数を特徴量とする場合

CASE3
システムコールの呼出された回数にエラーの呼び出された回数を引いたものを特徴量とする場合

CASE4
API の呼出された回数のみを特徴量とする場合

表 2 から見ると CASE3 のシステムコールの呼び出された回数にエラーの呼び出された回数を引いたものを特徴量とする場合に API の呼び出された回数も特徴量として取り

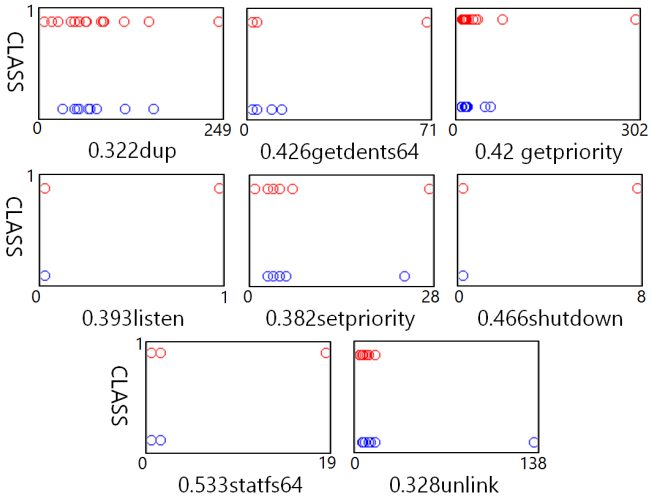

図 3　PCA による重みの大きいクラス別の特徴量（CASE3）
Fig. 3　using the PCA, feature value of the large-weighted by class.(CASE3)

表 2　CASE ごとに K-means 適用した結果
Table 2　Result of K-means clustering by each CASEs.

	API あり		API なし	
	PCA なし	PCA あり	PCA なし	PCA あり
CASE1	87.5000%	62.5000%	83.3333%	58.3333%
CASE2	79.1667%	58.3333%	79.1667%	62.5000%
CASE3	91.6667%	58.3333%	83.3333%	54.1667%

入れ，主成分分析を行わない場合が 91.6667%という一番良い結果を見せている．全体的に API を特徴としない場合より API を特徴にした場合が良い結果を見せていることから，既存研究の手法であるシステムコール呼び出し回数のみを特徴にする場合より精度が良くなったと考えられる．

また，主成分分析を用いて次元圧縮すると全体的に精度が落ちている．CASE3 の場合，402 次元から 15 次元に圧縮された．次の**図 3** に 15 次元の主成分の中で重みが大きい順で 8 個の特徴を選び，クラス別にその特徴量である呼び出された回数を示す．CLASS0（青）のクラスが正規アプリケーションでＣＬＡＳＳ１（赤）のクラスがマルウェアである．この 8 個の特徴はすべてシステムコールに属するもので，特徴名の左に付いてる数値は主成分の重みを表す．どれを見ても正規アプリケーションとマルウェアの呼び出された回数に差が見れない．つまり，重みの小さい特徴もクラスタリングに影響が大きいと考えられる．従って，主成分分析を行うと精度が落ちると考えられる．

5.2　情報漏えい検知

本節では API 呼び出し回数のみを特徴とした場合，K-means クラスタリングアルゴリズムを適用した結果をもとに情報漏えい検知の可能性について議論する．**表 3** に API 呼び出し回数のみを特徴とした場合にクラスタリング

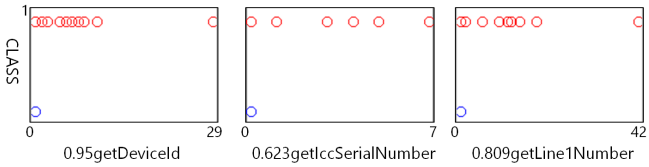

図 4　PCA による重みの大きいクラス別の特徴量（CASE4）
Fig. 4　Using the PCA, feature value of the large-weighted by class.(CASE4)

表 3　API 呼び出し回数のみを K-means 適用した結果
Table 3　Result of K-means that only takes in number of the API called times.

	PCA なし	PCA あり
CASE4	75.0000%	83.3333%

表 4　インスタンス別クラスター結果
Table 4　Result of cluster by each instance.

	CASE3（91.6667%）		CASE4（83.3333%）	
	Cluster0	Cluster1	Cluster0	Cluster1
REGULAR	7	2	9	0
MALWARE	0	15	4	11

を行った結果を示す．今回は主成分分析を行ったときがが行わないときより良い精度を見せている．CASE4 のデータに主成分分析を行った場合，次元を 3 つに圧縮できた．その 3 つの主成分で重みが大きい API のクラス別の呼び出された回数を次の**図 4** に示す．同様に，CLASS0（青）のクラスが正規アプリケーションでＣＬＡＳＳ１（赤）のクラスがマルウェアである．図 4 から見ると正規のアプリケーションはこういった API は全く呼び出していないが多くのマルウェアは呼び出していることがわかり，はっきりと差がつく．このことより，CASE4 の場合は主成分分析を行うと良い精度を見せたと考えられる．

さらに，これらの API が取得する情報は，API 名から推測できる．例えば，getDeviceId は端末識別番号（IMEI），getIccSerialNumber は SIM カードのシリアルナンバー，getLine1Number は電話番号といった個人情報である [18]．従って，CASE4 よりマルウェアに分類されたインスタンスはこういった情報を取得していると考えられる．CASE3 で API あり，PCA なしの場合と CASE4 の PCA ありの場合に正規アプリケーションとマルウェアが実際にどのクラスターに分類されたかを示す表を次の**表 4** に示す．どちらも Cluster0 が正規だとみられるクラスターで Cluster1 がマルウェアだとみられるクラスターである．CASE4 で Cluster1 のマルウェアインスタンス 11 個は上記の API を呼び出し情報を取得していて、漏えいさせていると考えられる．Cluster0 にある残りの 4 個のインスタンスは CASE3 での結果が正しいと考えると上記の情報に対しては漏えいしていないが，他の脅威を持つマルウェアだと考えること

ができる．しかし，実際にそうなのかは本実験ではわからないので，考察する必要がある．

5.3 考察

結果よりシステムコール呼び出し回数のみを特徴とするより API 呼び出しも特徴に含めた方が良いことがわかる．本実験では Strace よりシステムコールの統計情報を取得するとエラーが呼び出された回数の情報も入っていたため，様々な方法でシステムコールが呼び出された回数に取り込んだ．結果，システムコールにエラーの呼び出された回数を引くことが精度が一番良かった．つまり，あるシステムコールがエラーになると，そのシステムコールを実行するために呼び出して，エラーになることを非正常的に繰り返していたと考えられる．

これはエミュレータ上で実験を行った影響があるかも知れない．また，本実験ではマルウェアインスタンスを含めてたくさんのアプリケーションサンプルをダウンロードしたが，実際には 8 割りほどのアプリケーションをエミュレータでは動かすことができなかった．エミュレータ上で動くアプリケーションのみに対して著者本人が実験を行いデータを収集したため，偏ったデータセットになっている可能性が十分ある．インスタンス数が計 24 個で比較的少ないことも結果に影響を与えていると考えられる．

また，主成分分析以外のこの手法に合う次元圧縮を工夫する必要がある．本実験では計 402 個の特徴を取っているが，実際にはアプリケーションインスタンスが使う特徴は大きく変わらなく，半分ほどの特徴は一度も呼び出されてないものが多い．そういった特徴を圧縮するためにも主成分分析を用いたが，特徴に重みを付けるだけでは分類の精度は上がらないことがわかった．特徴が多いと次元が高くなりバイアスがかかることなる．主成分分析以外の方法を用いて次元圧縮を工夫する必要がある．

本実験では TelephonyManager クラスの IPhoneSubInfo に定義されている API 呼び出し回数を実験対象としたため，計 8 個の情報漏えいに検知できる．システムコールでは特定できない機微な情報を呼び出す API をもっと記録し，統計情報を取得することで情報漏えい検知の範囲は広くなると見込んでいる．そして，マルウェアの分類にも有用な特徴になると考えられる．API をシステムコールと同様に特徴とするだけではなく，表 4 のように比較を行うことよりマルウェアがどのような情報を漏えいしているか検知可能となると考えられる．しかし，CASE4 で Cluster0 の 4 個のマルウェアインスタンス情報漏えいしていないクラスターに分類されているが，実際にそうであるかを確かめる必要がある．今回の 8 個の API はシステムコールでは特定できず，機微な情報を呼び出すことはわかった．しかし，必ずその API 利用せずには目的情報を呼び出せないことはわかっていないため，このことを確認しないと Cluster0 の 4 個のマルウェアインスタンスが本当にこの 8 個の API の取得情報は漏えいしていないと限れない．このことは今後の課題とする．

6. おわりに

6.1 実験のまとめ

マルウェアは主に非公式マーケットから拡散していき，公式マーケットでもマルウェアが検出されている [4]．アプリケーションの開発者以外の第 3 者であるアプリケーションマーケットの運営者側によるマルウェア検知が重要である．本実験では API 呼出しを記録するためには，AndroidOS のフレームワークを改変しカスタマイズする必要があるが，マーケットの運営者側が利用すると考えると問題にならないと考えられる．元々，Android の一般ユーザがアプリケーションをダウンロードする際に，潜在的な脅威を知ることが難しいことから，ユーザと開発者以外の第 3 者がアプリケーションの解析を行う必要があることが前提しているので，本実験の手法は当てはまると考えられる．

アプリケーションマーケットの運営者側がこの手法を利用することを想定すると，マーケットに登録要請するアプリケーションの多くは流通してないため，新種・亜種が混入していることが多い．従って，静的解析のパターンマッチング方式を使うアンチウイルスソフトやスキャナではパターン生成が追い付いていけず，マルウェアを見逃してしまう可能性が高居。また，新種・亜種に対しては教師データの作成が難しいため教師有り学習を用いることもできない．従って，本実験では大量の新種・亜種に対応するために教師なし学習の K-means クラスタリングアルゴリズムを用いることにした．

結果，既存研究の手法であるシステムコールの呼び出された回数のみを特徴量として分類するより API の呼び出された回数も取り込んで分類する方が良い精度を見せた．また，API 呼び出し回数のみを特徴として分類することより情報漏えいの検知も確認できた．しかし，今回の 8 個の API はシステムコールでは特定できず，機微な情報を呼び出すことはわかったが，必ずその API を利用せずに目的情報を呼び出せないないことはわかっていないため，このことを確認する必要がある．

6.2 今後の課題

エミュレータ上で実験を進めるには不具合が多かったため，実機で実験を進めることを今後の課題とする．これはインスタンス数の増加に繋がり，より正確なデータベース作成に繋がると思う．また，今回の実験ではアプリケーションインスタンスを収集したのが著者本人でマルウェアを特定することができたため，実験は著者本人以外の第 3 者に委託してデータを収集するか，アプリケーションイン

スタンスを第 3 者に委託して収集してもらうようにする．また，全システムコールと API を特徴とすると 400 次元の超える高次元になってしまう．しかし，次元を圧縮する手法の一つである主成分分析でマルウェアを分類するには本実験では有効ではなかったため，主成分分析以外の次元圧縮に工夫をする必要がある．クラスタリングにも予めいくつかのラベル付きデータを用意するなどの半教師あり学習なども工夫すると精度が上がると考えられる．

API はマルウェアの分類にも有効で，情報漏えい検知にも活用できたため記録する API の数を増やすと精度が上がると見込んでいる．従って，システムコールでは特定できないが，機微な情報を呼び出す API を対象として記録を増やすことを今後の課題とする．また，逆にそういった API を呼び出さずに目的情報を呼び出す別の方法が存在するか調べることが必要になると思う．もし方法が存在しなければ確実に情報漏えいを検知できると考えられる．

謝辞　本研究を進めるにあたり，ご多忙の中に有益な助言を頂いた長崎県立大学 穴田啓晃准教授，大日本印刷 梶原直也様，そして研究に協力頂いた九州大学櫻井研究室の皆様に心から感謝いたします．また，本研究は JSPS 科研費 26330169 の助成を受けたものです。

参考文献

[1] I. Burguera, U. Zurutuza, S. Nadjm-Tehrani, “Crowdroid: behavior-based malware detection system for Android”, *In Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices.* pp.15-26, 2011.

[2] Y. Nishimoto, N. Kajiwara, S. Matsumoto, Y. Hori, K. Sakurai, “Detection of Android API call using logging mechanism within Android Framework”, *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering.* Vol.127, pp.393-404, 2013.

[3] G DATA mobile malware report, `https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/G_DATA_MobileMWR_Q2_2015_EN.pdf`.

[4] Brain Test re-emerges: 13 apps found in Google Play, `https://blog.lookout.com/blog/2016/01/06/brain-test-re-emerges/`. [Accessed in February 2016]

[5] T. Isohara, K. Takemori, A. Kubota, “Kernel-based Behavior Analysis for Android Malware Detection”, *In Proceedings of the 7th International Conference on Computational Intelligence and Security.* pp.1011-1015, 2011.

[6] 笠間貴弘, 吉岡克成, 井上大介, 松本勉, “実行毎の挙動の差異に基づくマルウェア検知手法の提案”, コンピュータセキュリティシンポジウム *2011* 論文集. pp.726-731, 2011.

[7] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. Gaur, M. Conti, R. Muttukrishnan, “Android security: a survey of issues, malware penetration and defenses”, *Communications Surveys Tutorials, IEEE.* Vol.17, pp.998-1022, 2015.

[8] W. Fan, Y. Liu, B. Tang, “An API calls monitoring-based method for effectively detecting malicious repackaged applications”, *International Journal of Security and Its Applications.* Vol.9, pp.221-230, 2015.

[9] S. Zou, J. Zhang, X. Lin, “An effective behavior-based Android malware detection system”, *Security and Communication Networks.* Vol.8, pp.2079-2089, 2015.

[10] K. J. Abela, J. R. Delas Alas, D. K. Angeles, R. J. Tolentino, M. A. Gomez, “An Automated Malware Detection System for Android using Behavior-based Analysis AMDA”, *International Journal of Cyber-Security and Digital Forensics.* pp.1-11, 2013.

[11] Android open source project. Downloading the source, `http://source.android.com/source/downloading.html`. [Accessed in December 2015]

[12] Linux Programmer’s Manual, `http://man7.org/linux/man-pages/man2/syscalls.2.html`. [Accessed in January 2016]

[13] S. Sheen, R. Anitha, V. Natarajan, “Android based malware detection using a multifeature collaborative decision fusion approach”, *Neurocomputing.* Vol.151, pp.905-912, 2015.

[14] Y. Aafer, W. Du, H. Yin, “DroidAPIMiner: Mining API-level features for robust malware detection in Android”, *Security and Privacy in Communication Networks.* Vol.127, pp.86-103, 2013.

[15] V. M. Afonso, M. E. de Amorim, A. R. A. Gregio, G. B. Junquera, P. L. de Geus, “Identifying Android malware using dynamically obtained features”, *Journal of Computer Virology and Hacking Techniques.* Vol.11, pp9-17, 2015.

[16] P. P. K. Chan, W. Song, “Static detection of Android malware by using permissions and API calls”, *Machine Learning and Cybernetics.* Vol.1, pp82-87, 2014.

[17] Android developers. dashboards(Platform Versions), `https://developer.android.com/intl/en/about/dashboards/index.html`. [Accessed in December 2015]

[18] `http://developer.android.com/reference/android/telephony/TelephonyManager.html` [Accessed in January 2016]

[19] A. Sharma, S. K. Dash, “Mining API Calls and Permissions for Android Malware Detection”, *Cryptology and Network Security.* Vol.8813, pp.191-205, 2014.

[20] Android malware data sets, `http://cgi.cs.indiana.edu/~nhusted/dokuwiki/doku.php?id=datasets`. [Accessed in January 2016]

[21] `https://www.virustotal.com/`. [Accessed in January 2016]

[22] D. Wu, C. Mao, T. Wei, H. Lee, K. Wu, “DroidMat: Android Malware Detection through Manifest and API Calls Tracing”, *in Proceedings of 7th Asia Joint Conference on Information Security.* pp.62-69, 2012.