動的解析を用いたAndroidにおける端末情報の取得検知手法

西本祐揮[†1]　堀良彰[†2]　櫻井幸一[†2]

Android プラットフォームを用いたスマートフォンの普及に伴い，Android を標的としたマルウェアが作られるようになった．しかしユーザがアプリケーションに潜在する脅威を把握して，アプリケーションの利用可否を判断することは難しい．本研究では，端末情報取得 API の実行時に遠隔手続き呼出しを用いて，情報取得が行われることに着目した．Android Binder による遠隔手続き呼出しにおける手続き実行時に Android のログ出力 API である Log.v メソッドを挿入することで端末情報取得 API にかかる呼出しを記録する手法を考案した．さらに，本考案手法を試験実装し，Android エミュレータ上で端末情報を取得するアプリケーションを動かし，端末情報取得 API の呼出し挙動の記録を実証的に確認した．

Detection of Terminal Information Access in Android by Dynamic Analysis

Yuuki Nishimoto,[†1] Yoshiaki Hori[†2] and Kouichi Sakurai[†2]

Android based smartphones have become popular. Accordingly, many malwares are developed. The malwares target information leaked from Android. However, it is difficult for users to judge the availability of application by understanding the potential threats in the application. In this paper, we focus on acquisition of information by using a remote procedure call when we run the API to acquire information of terminals. We design a system to record calls that are concerned the API by inserting Log.v methods. We execute our method, and confirm empirically the record of the call behavior of the API to acquire information of terminals.

1. はじめに

近年，Android プラットフォームを用いたスマートフォンが普及してきている．それとともにマルウェアと呼ばれる悪意あるアプリケーションも数多く作られている．Android をターゲットにしたマルウェアには情報漏洩を行うものも多く，Android ユーザの個人情報の漏洩が大きな問題になっている．しかし，ユーザがアプリケーションに潜在する脅威を把握して，アプリケーションの利用可否を判断することは実際には難しい．そこで，ユーザに代わってアプリケーション開発者とユーザ以外の第三者が予めアプリケーションを検査することで、ユーザが誤ってマルウェアを含むアプリケーションをインストールすることを防止する対策手法に着目する．

マルウェア検知の手法には静的解析と動的解析がある．静的解析ではマルウェアの亜種が大量発生したときに見逃しが増える可能性がある[1)]．そこで今回は動的解析について研究した．Android にはアプリケーションが使用したシステムコールを監視する，strace という Linux のデバックユーティリティが存在する．strace を用いて取得したシステムコールを解析することでマルウェア検知を行う手法がある．この手法ではカーネルのサービスを利用する挙動は検知できる．しかし，カーネルのサービスを利用しない挙動ではシステムコールが発行されず，その挙動を検知することができないという課題がある．

本研究では，端末情報取得 API を実行した際に，遠隔手続き呼出しを用いて情報取得が行われることに着目した．Android Binder による遠隔手続き呼出しにおける手続き実行時に Android のログ出力 API である Log.v メソッドを挿入することで端末情報取得 API にかかる呼出しを記録する手法を考案した．この手法による実行記録は呼び出し側においてAPI の改変が行われる場合でも回避することはできない．さらに，本考案手法を試験実装し，Android エミュレータ上で端末情報を取得するアプリケーションを動かし，端末情報取得 API の呼出し挙動の記録を実証的に確認した．

†1 九州大学工学部電気情報工学科
Department of Electrical Engineering and Computer Science, School of Engineering, Kyushu University
†2 九州大学大学院システム情報科学研究院情報学部門
Department of Informatics, Faculty of Information Science and Electrical Engineering, Kyushu University

2. Android

Android とは，スマートフォンやタブレット PC などの携帯情報端末を主なターゲットとして開発されたプラットフォームである．Android アプリケーションは Dalvik 仮想マシン上で実行される．Dalvik 仮想マシンとは Android で用いられるレジスタベースの仮想マシンである．1 つのアプリケーションを実行するために 1 つの Dalvik 仮想マシンを立ち上げることによって，アプリケーション同士を独立に実行させている．アプリケーションをインストールするときに，ユーザがパーミッションを承認することでサンドボックス機構を逸脱し，ほかのアプリケーションとの連携を取る，他のアプリケーションが作成したファイルにアクセスする，などということが可能になる[2)]．

端末中の情報や機能などに対するアクセスを制御するためのパーミッションは，それぞれのパーミッションで許可される情報や機能などを複数組み合わせることで，ユーザーの意図しない挙動を行うことがある．Android について深い知識を持たないユーザでは，このような複数のパーミッションの組み合わせから発生する危険性を予測することは難しい．

2.1 Android の構成

Android は Linux ベースのモバイル用オープンソース・オペレーティングシステム，ミドルウェア，主要なアプリケーションからなるソフトウェアスタック（集合）パッケージから構成されている．ユーザーが使用するアプリケーションは AndroidOS の中でも一番上の階層にあたる．アプリケーションは Android Market で公開，配布が可能であり，世界中の誰でも自由に利用することができる．

Android のアプリケーションフレームワークでは，Android アプリケーションの開発者が自由に使えるものとして API を提供している．これによってアプリケーション開発者はフレームワーク層以下の複雑な手続きについて知ることなく，その API が返す結果を利用できる．

Android アプリケーション

Android アプリケーション・ソフトウェアは Java 言語で記述されているものが多い．実行ファイルである DEX ファイルや，画像ファイル，パーミッションを記した XML ファイルなどを圧縮した Android パッケージは，apk という拡張子を持つ．ユーザはこのパッケージを Android Market からダウンロードしてインストールするか，もしくはそのパッケージが保存されている PC からコマンドプロンプトを利用して直接インストールする．アプリケーションに関する情報は 1 つの apk ファイルにすべて含まれているため，apk ファイル全体で 1 つのアプリケーションパッケージを構成する．AndroidOS ではアプリケーション同士が干渉を起こして不具合を発生させることがないように，以下の方法でアプリケーション同士の独立を保っている．

- **アプリケーション実行とプロセス**
  Android アプリケーションは，そのアプリケーション個別の Linux プロセスで実行されている．Linux プロセスは，アプリケーションを実行するときにプロセスが開始される．しかしプロセスが終了するのはアプリケーションが終了した時ではなく，アプリケーションが終了したあとに，他のアプリケーションからシステムリソースを要求されたときである．
- **プロセスごとに割り当てられた Dalvik 仮想マシン**
  Android ではプロセスごとに専用の Dalvik 仮想マシンが割り当てられている．これによって一つのアプリケーションはその他のアプリケーションから独立して実行される．
- **アプリケーションごとに割り当てられた固有の Linux ユーザ ID**
  Android アプリケーションはインストールの際に，各アプリケーション固有のユーザ ID が割り当てられる．その割り当てられたユーザ ID がそのアプリケーションの所有者となり，プロセスを実行する．そしてそのアプリケーションが作成したファイルは，基本的に他のユーザ ID を持つアプリケーションからは読み込めないように設定されている．このため，別のユーザ ID を持つアプリケーションから，あるアプリケーションが作成したファイルを自由に読み込むことはできない．

2.2 Dalvik 仮想マシン

Dalvik 仮想マシンとは低メモリ環境に最適化された仮想マシンである．主な働きとして各アプリケーションごとのプロセス間の分離，メモリ使用量を最小化するように最適化された Dalvik 実行可能フォーマット (DEX) ファイルの実行，複数の仮想マシンインスタンスの効率的な同時実行などがあげられる．Android では，アプリケーションを実行する際，Dalvik 仮想マシンのサンドボックス上で実行する．Android アプリケーションが Java 言語で書かれることが多いため誤解されることがあるが，Dalvik 仮想マシンは.class ファイルを直接実行するものではない．Android アプリケーションを作成するときは，Java 言語で書いたソースコードを.class ファイルにコンパイルしてから dx という変換ツールを用いて DEX ファイルに自動変換し，それを実行している．

ハードウェアリソースが限られている携帯端末環境の中で Dalvik 仮想マシンを利用する理由は，2 つあげられる．1 つはアプリケーションの汎用化である．仮想マシンを利用して

アプリケーションを動かすことで，CPU の種類や数を考慮することなくアプリケーションを開発できる．2 つ目に Dalvik 仮想マシンがレジスタマシンであることがあげられる[3]．スタックマシンでは命令数が多いことからインタプリタでのパフォーマンスが低下する．これはインタプリタが命令そのものの処理よりも，命令ごとに発生するオーバーヘッドの方が大きいためである[4]．

通常利用できない情報や機能などにアクセスしてそれらを利用する場合，パーミッションを利用する．Android アプリケーションに含まれる AndroidManifest.xml というファイルの中に，必要とされるパーミッションは記載されている．この中に記載されているパーミッションを，パッケージマネージャと呼ばれる Android プラットフォームのアプリケーション管理モジュールを介して，インストール時にユーザが承認する．パーミッションを承認されたアプリケーションは許可された範囲内で自由にサンドボックスの外にある情報や機能などにアクセスすることができるようになる．パーミッションが必要となる具体的な例としてはインターネットアクセス，ユーザのアドレス帳データの読み込みと書き込み，WiFi や GPS などを用いた位置情報の取得といった特定の処理があげられる．

2.3 Binder

Binder とはプロセス間で通信する機能を提供する Android 固有のドライバのことである．各プロセスは同じアプリケーション内であっても，それぞれ独立した領域で動いている．また，そのアプリケーションの中でもアクティビティとサービスが，異なるプロセスでそれぞれ動いている可能性もある．これら異なるプロセス間で情報をやり取りするときに Binder ドライバを利用する．この時，実際の通信はカーネルより上に位置するフレームワーク層で制御する．Binder はユーザが直接利用することはないが，プロセス間通信において重要な役割を担っている．

2.4 AIDL (Android Interface Definition Language)

Android 上では通常，1 つのプロセスは別のプロセスのメモリにアクセスすることはできない．そこで別プロセスからデータを得たい場合プロセス間通信を行わなければならない．AIDL(Android Interface Definition Language) とは，コードを生成するために利用されるインターフェイス定義言語のことである[5]．このコードは Android 上で 2 つのプロセスが Binder を用いたプロセス間通信を利用して，データのやり取りを行うことを可能にするためのものである．

3. Android API

API とは Application Programming Interface の略称であり，アプリケーションから OS やプログラミング言語に用意されているライブラリなどの機能にアクセスするためのインターフェースのことである．API で提供されている機能に関しては新たに開発する必要がなくなるので，API を利用することでアプリケーションなどの開発を効率的に進めることができるようになる．

Android API は AndroidOS 用のアプリケーションを作成するときに使用できる命令や関数などの集合である．多くのアプリケーションで利用される機能は Android のアプリケーションフレームワークの中でまとめて提供されている．AndroidAPI は多数のパッケージから構成されており，そのパッケージ一つ一つの中で API のメソッドを含んだクラスやインターフェースなどが定義されている．AndroidAPI にも OS の基盤となる機能やスクリーンに対する描画機能など，WindowsOS が提供する機能をアプリケーションなどのプログラムから利用するためのインターフェースである，WindowsAPI と共通する機能を提供するものもある．しかし AndroidAPI には WindowsAPI にはない API として，電話番号や契約者固有 ID など，基本的な端末の情報を管理する TelephonyManager クラスの API が存在する．TelephonyManager クラスには端末識別番号を取得する getDeviceId() やユーザの電話番号を取得する getLine1Number()，ユーザの契約者固有 ID を取得する getSubscriberId() など，個人情報にかかわる API が複数定義されている．

4. 既存の Android マルウェアとその検知手法

アプリケーションの解析技術には静的解析と動的解析がある．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静的解析とアプリケーションを実行して挙動を見る動的解析にはそれぞれメリット，デメリットがあり，一概にどちらが良いとは言えない．この章では静的解析と動的解析の概要と，それぞれの手法を利用したマルウェアの検知手法について論じる．

4.1 Android マルウェア

マルウェアとは，端末内部の個人情報を漏洩させたり，端末内部のデータを破壊させたりするなど，悪意ある挙動を行うアプリケーションのことを指す．一口にマルウェアといっても，どのような悪意ある挙動を行うかに基づいて分類される．マルウェアはターゲットの多い環境に合わせて作られることが多い．そのため今までは利用者の多い WindowsOS

をターゲットにしたマルウェアが，マルウェア全体でも多くの割合を占めてきた．しかし，2010 年後半から AndroidOS が急速に普及するにしたがってその利用者も急速に増加した．そのためマルウェアの開発者たちも，営利目的で AndroidOS を次なる標的として狙うようになった．G Data Malware Report –Half-yearly report January - June 2011–[6] によると，Android を中心としたスマートフォンをターゲットとしたマルウェアは 2010 年下半期から 2011 年上半期にかけて，55 件から 803 件と 13.8 倍も増加している．絶対数では Windows を狙ったマルウェアとは比べ物にならないくらい低いが，Android が端末内に保存している情報が，電話番号や契約者 ID などという重要な個人情報であることを考えると，他の OS と比べてセキュリティ面でのリスクが高くなると考えられる．これらのことから，AndroidOS におけるマルウェア対策を含めたセキュリティの充実が急務になっているにもかかわらず，現在セキュリティが充実した状態であるとは言えない．Android 上で直面するマルウェアの脅威のほとんどは，Google の正規の Android Market ではない第三者のマーケットプレイスから入手したマルウェアをインストールすることによる感染である．

Android のマルウェアには次のような特徴がある．

- マルウェア単体でアプリケーションの体を成すもの
  アプリケーションそのものがマルウェアであるタイプを指す．
- 何らかのアプリケーションの裏で動くマルウェア
  実行したら表面上は正規のアプリケーションのようにふるまうが，サービス機能を利用して裏で悪意ある動作を行うタイプを指す．感染してもユーザーは気が付きにくい．これには正規のアプリケーションを改造してマルウェアを組み込んだものがある．Android Market などで有料で売られているアプリケーションを無料で配布しているサイトからダウンロードすると，感染していることが多い．このマルウェアには 2 種類ある．
  – 正規のアプリケーションのコードにマルウェアコードを書き加えたもの
    マルウェアのコードが正規のアプリケーションのコードの中に挿入されているだけなので静的解析でコード解析を行えば発見できる．
  – 正規のアプリケーションのコードを悪意ある行動をするように改変したもの
    正規のアプリケーションのプログラムの中に悪意あるコードが混ざっているので静的解析では発見が難しい．

4.2 静的解析

静的解析とは，アプリケーションを逆コンパイルすることで実行ファイルを実行することなく解析を行うソフトウェアの解析手法のことである．主にソースコードを解析するときに利用されることが多い．

実際にアプリケーションを動かすことなく解析を行うため，マルウェアの被害にあう前にその潜在的な脅威を検知できる．その一方でアプリケーションのソースコードが難読化された場合や，アプリケーションの中で外部サーバと連携を行う機能を用いて，攻撃用のコードをアプリケーションコードの外部に設置された場合などは検知できない可能性が高くなる[7]．

4.3 動的解析

動的解析とは，被検査アプリケーションを実際に動かしてみることで，そのアプリケーションがどのような挙動をしているのかを確認するプログラム解析手法である．静的解析と異なり実際にアプリケーションを動かして，その挙動を基に検査を行うため，アプリケーションが難読化されていた場合や，アプリケーションコードの外部に攻撃用のコードを置かれた場合などでもマルウェアとして検知することができる．しかしこの手法も完全な検知手法ではなく，実際にアプリケーションを動かして得たデータが，そのアプリケーションの取りうるすべての挙動であるかどうかの判別が難しいという欠点が存在する．例えばある特定の日時にのみ情報を漏洩するマルウェアを検査した場合，その特定の日時以外でいくら検査を重ねてもそのアプリケーションがマルウェアであるという結果は出てこない．

5. Logcat を用いたプロセス動作の記録手法の考案

5.1 プロセス動作の記録手法

strace とはプログラムが発行するシステムコールを監視するためのデバッグユーティリティである．この strace を用いたプロセスの動作の記録手法において，システムコールを発行しない TelephonyManager クラスの API を用いたプログラムから，その API に関するシステムコールを得ることはできない．これは TelephonyManager クラスの API を利用して情報の呼び出しを行ったとき，カーネルのサービスを利用することなく目的の情報を呼び出しているためである．

AndroidOS のアプリケーションフレームワークの中に Log.v メソッドを挿入して，ログを出力させるように AndroidOS のアプリケーションフレームワークを改変する．そしてアプリケーションが API を介して端末内部の情報を取得するときにそのログを記録し，それを基に情報の呼び出し検知を行う手法を用いる．この手法は他の OS でも用いられており，UNIX 系 OS では syslog という機能を，Windows ではイベントログという機能を用いてそれぞれログを記録する．

今回調査している AndroidOS のアプリケーションフレームワークには logcat と呼ばれ

表 1 IPhoneSubInfo.aidl で定義された API
Table 1 API defined in IPhoneSubInfo.aidl

API	取得する情報
getDeviceId()	IMEI(端末識別番号)
getDeviceSvn()	デバイスの
(getDeviceSoftwareVersion() 内部のメソッド)	ソフトウェアバージョン
getSubscriberId()	契約者固有 ID
getIccSerialNumber()	SIM カードの
(getSimSerialNumber() 内部のメソッド)	シリアルナンバー
getLine1Number()	電話番号
getLine1AlphaTag()	アルファベット識別子
getVoiceMailNumber()	ボイスメール番号
getCompleteVoiceMailNumber()	完全ボイスメール番号
	ボイスメール
getVoiceMailAlphaTag()	アルファベット識別子

る機能を用いてログを記録する API が用意されている．今回の実験では Android フレームワーク内の Java 言語を用いている層で，android.util パッケージで定義されている Log クラスの Log.v() メソッドから出力されるログを収集し分析調査した．

ログを出力させてどのメソッドが呼ばれたかを確認する手法は，他の OS でも行われている一般的な手法である．Android において 1 つのアプリケーションを実行するときは，必ず独立した 1 つの Dalvik 仮想マシン上で実行される．よってほかのプロセスと通信を行うときは直接行うことはできず，他のプロセスとの通信を行いたい場合は Binder という Android 固有のドライバを利用しなくてはいけない．そこでこのプロセス間通信を行うプログラム中にログを出力するプログラムを置き，そこから得られる情報を基に，Dalvik 仮想マシン上で動いているアプリケーションが API を呼び出したときに，それを検知・特定しようというのが今回の提案で新しい部分である．この技術を用いればそれぞれの情報を直接読み込む部分を探索することなく，AIDL で書かれた.aidl ファイルを見ることで呼び出された情報を確認できる．今回の実験ではアプリケーションを実行させて，得られたログの中から特定のログが出力された時にそのログから得られる情報を利用して，アプリケーション実行中に呼び出された API を特定する．

本研究では電話番号や契約者 ID など，重要度の高い情報を扱う IPhoneSubInfo.aidl ファイルに含まれる API のみを対象に実験を行った．その具体的な手法として，まず IPhoneSubInfo.java に android.util パッケージの Log クラスをインポートする．その後，onTransact メソッドにログを出力するプログラムを挿入する．その後 OS を再コンパイルする．その OS に複数の API を呼び出すアプリケーションをインストールし実行する．得られたログの中から，code と呼ばれる onTransact 内で使われている変数に注目する．**表 1** に IPhoneSubInfo.aidl ファイルに含まれる API と各 API の呼び出す情報の対応表を示す．今回の実験ではこの中から getLine1AlphaTag() と getCompleteVoiceMailNumber() の 2 つに関しては実験を行なっていない．その理由は以下の 2 つである．

- TelephonyManager.java 内にプログラミングされているにも関わらず，Android Developer のサイト内の TelephonyManager クラスのメソッドとして表記されていないこと
- アプリケーション内で TelephonyManager クラスのメソッドとして使おうとすると，TelephonyManager 内で未定義であるというエラーが出ること

以上のことから今回は先述の 2 つを除く 7 つの API に関して実験を行った．

5.2 実験概要

今回の実験ではアプリケーションを逆コンパイルしてソースコードを解析する静的解析ではなく，実際にアプリを動かしてそのアプリに関するログを取ることで情報の漏洩を検知する動的解析が目的となる．先述したように近年のマルウェアはアンチマルウェアソフトなどから検知されるのを防ぐため，解析を困難にするために自身を難読化していたり，webkit を利用した外部サーバーとの連携によって情報漏洩を行ったりといった工夫を凝らしてきている．今回動的解析を選択した理由は，動的解析ならばそのような静的解析では対処しきれない状況にも対応できるためである．

strace を用いたプロセスの動作の記録手法から TelephonyManager の API で呼び出される個人情報は，カーネルから呼び出すのではなく，他の情報管理プロセスから受け渡されることが予想される．そこで提案手法ではプロセス間通信で重要な役割を持つ Binder ドライバに着目した．本研究では IPhoneSubInfo.aidl に記述されている API が呼び出されたときに，それを検知する実験を行った．これらの API が呼ばれた時にのみ呼ばれる IPhoneSubInfo クラスの onTransact メソッドに，ログを出力するメソッドである Log.v を挿入して，得られたログを基に呼び出された API の特定を試みた．IPhoneSubInfo クラスの onTransact メソッドには code という引数が存在する．onTransact メソッドはこの code 変数の値で，IPhoneSubInfo.aidl 内で定義されているどのメソッドから情報呼び出しがあったのかを判断している．ゆえにこの IPhoneSubInfo クラスの onTransact メソッドが呼ばれたことと，その中の code 変数を判定することでどのメソッドが呼ばれたのかを特定できると考えた．

5.3 提案手法

図 1 に示すように IPhoneSubInfo.java 内部にログを出力するプログラムを挿入する．ログを挿入する場所は以下のような条件を考慮して決定した．

- 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はないこと
  TelephonyManager クラスの getDeviceId() メソッドはアプリケーションと同じプロセスで実行されている．このようなメソッドは，アプリケーションを開発するときにアプリケーション内にライブラリとして組み込むことができる．TelephonyManager クラス内の getDeviceId() メソッドにログを出力するプログラムを挿入した場合，マルウェア開発者側がアプリケーション内に，プログラムとして同じ働きをするメソッドを定義してそのメソッドを実行した場合，ログの出力は行われないまま情報の呼び出しが行われてしまう．ゆえにアプリケーションのプロセスと違うプロセスで実行されるメソッドにログを出力するプログラムを挿入することは，きわめて重要なことである．
- メソッドを呼び出した API が特定できること
  ログを出力するプログラムを挿入したメソッドからログが出力されても，そのログからアプリケーションがどの API を呼んだのかを確認できなければこの手法は成り立たない．
- できるだけ多くの API の呼び出し検知に利用できるメソッドであること
  ログを出力するプログラムを挿入する場所に直接何らかの情報の値を返す場所を指定してしまうと，その情報しか監視できなくなってしまう．

以上 3 つの点を踏まえて検討した結果，IPhoneSubInfo クラスの onTransact メソッドが適当であるという結論に至った．その理由として，アプリケーションと同プロセスで実行されていないこと，IPhoneSubInfo.aidl ファイルでメソッドが定義された順番と code 変数の値が一致していることから，どの API が呼び出されたか判断できること，IPhoneSubInfo.aidl 内で定義された 9 つの API について検査できることがあげられる．

5.3.1 利　　点

この手法の利点として実行時にシステムコールが発行されない API を検知できることがあげられる．既存研究では動的解析を用いるときにシステムコールのログを取り，マルウェアの検知を行っている．しかしこの手法では実行時にシステムコールが発行されない API を検知することは難しい．しかし提案手法では API が呼び出されたときに利用されるプロセス間通信に着目し，そのメソッドが呼び出されたときにログを出力するプログラムを挿入したため，システムコールによらない情報呼び出しの検知が可能になる．

5.3.2 欠　　点

この手法の欠点としてリアルタイム性がないということがあげられる．提案手法は，アプリケーションを実行したときに出力されるログをアプリケーションの実行終了後に解析しているので，被検査アプリケーションがマルウェアであった場合，情報漏洩が起きてしまっている可能性がある．

5.4 実験手順

(1) AndroidOS の入手
AndroidOS はオープンソースであるので AndroidDeveloper というサイトにソースコードのダウンロードのやり方が紹介してある．その手順通りに進めていき，AndroidOS を入手する．

(2) 入手したソースコードのビルド
AndroidOS のダウンロードが終了したあと，make コマンドを実行してビルドする．ここで IPhoneSubInfo.aidl ファイルから自動的に IPhoneSubInfo.java ファイルが作成される．java ファイルの内容は自動的に作成される．

(3) ログを出力するプログラムの挿入
図 1 には IPhoneSubInfo.java 中に挿入した，ログを出力するためのプログラムを示している．このプログラムは Logcat ビューで見ることができるログを出力するものである．Log.v は詳細メッセージのログを出力するタイプである．ほかにエラーのログを出力する Log.e，警告のログを出力する Log.w，情報のログを出力する Log.i，デバックメッセージのログを出力する Log.d がある．基本的に使い方は同じで，1 番目の引数にタグを示す文字列を，2 番目の引数にログとして出力したい文字列をそれぞれ指定する．この 5 つの違いは用途にみられ，あとでログを確認するときに便利なように使い分けられている．このログが出力するメッセージの内容は，IPhoneSubInfo.onTransact という文字列と，code という変数に何が入っているかという情報である．

(4) 再ビルド
IPhoneSubInfo.java ファイルを書き換えて保存した後，再度ビルドを行う．

(5) エミュレータ上で実行
今回の実験はすべてエミュレータ上で行った．

(6) ログの参照
DDMS(Dalvik Debug Monitor Service) ツールを利用して Android が収集したログ

```
@Override public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply,
int flags) throws android.os.RemoteException
{
Log.v("EXTEST","IPhoneSubInfo.onTransact, code:" + code);
switch (code)
{
case INTERFACE_TRANSACTION:
{
reply.writeString(DESCRIPTOR);
return true;
}
case TRANSACTION_getDeviceId:
{
data.enforceInterface(DESCRIPTOR);
java.lang.String _result = this.getDeviceId();
reply.writeNoException();
reply.writeString(_result);
return true;
}
・・・・・・
・・・・・・・・・・・・
```

ログを出力するプログラム

図 1 ログを出力するプログラムを挿入した IPhoneSubInfo.java のプログラム
Fig. 1 Source code inserted a program that prints logs in IPhoneSubInfo.java

を参照する．ここで収集したログをもとに結果の記述と考察を行った．

5.5 結　　果

出力されたログから呼び出された情報を特定することができた．**表 2** に実験で使用した API と code 変数の対応表を示す．これは IPhoneSubInfo.aidl ファイル内でそれぞれのメソッドが定義されている順番と対応する．このことから AndroidOS のソースコードからそれぞれのメソッドに対応する code 変数を知ることができることがわかった．

図 2 には getDeviceId() メソッドを実行したときに出力されるログを示す．今回に実験ではメソッドの関係をわかりやすくするため，提案手法の中で説明しているログ以外にも，主要なメソッドにログを出力するプログラムを挿入している．それぞれの図の中で赤い下線が引いてある部分が，本研究で提案した手法の中で必要な情報を出力するログである．

5.6 考　　察

今回の実験から，本論文の提案手法でアプリケーションが呼び出した API の検知と特定が可能であることがわかった．今回の実験では IPhoneSubInfo.aidl 内で定義された API の

表 2 実験で使用した API と code 変数の対応表
Table 2 Correspondence table API used experiment and code variable

code	API
1	getDeviceId() getDeviceSvn()
2	(getDeviceSoftwareVersion() 内部のメソッド)
3	getSubscriberId() getIccSerialNumber()
4	(getSimSerialNumber() 内部のメソッド)
5	getLine1Number()
6	getLine1AlphaTag()
7	getVoiceMailNumber()
8	getCompleteVoiceMailNumber()
9	getVoiceMailAlphaTag()

```
PID  Message
335 getDeviceId_begin
335 TelephonyManager.getDeviceId
147 Binder.execTransact_in
147 Binder.execTransact_try_in
147 IPhoneSubInfo.onTransact, code:1
147 PhoneSubInfoProxy.getDeviceId
147 PhoneSubInfo.getDeviceId-permission_before
147 ContextWrapper.enforceCallingOrSelfPermission
147 ContextImpl.enforceCallingOrSelfPermission
147 ContextImpl.checkCallingOrSelfPermission
74  Binder.execTransact_in
74  Binder.execTransact_try_in
74  ActivityManagerService.onTransact
74  ActivityManagerNative.onTransact
74  Binder.execTransact_try_out
74  Binder.execTransact_out
147 PhoneSubInfo.getDeviceId-permission_after
147 GSMPhone.getDeviceId:000000000000000
147 Binder.execTransact_try_out
147 Binder.execTransact_out
335 getDeviceId_done
```

IPhoneSubInfo.javaの
onTransactメソッド内に
挿入したプログラムから
出力されたログ

図 2 getDeviceId() メソッドを実行したときに出力されるログ
Fig. 2 Logs that are output when we run getDeviceId() method

みに注目したため，IPhoneSubInfo クラスの onTransact メソッドにログを出力するプログラムを挿入した．今回の実験では取り上げなかった API についても，プロセス間通信を行うものならば.aidl ファイル内で定義されていると考えられるので，その aidl ファイルを発見し本実験と同様の実験を行うことでその API の挙動を検知することができると考えられる．

6. 終わりに

本論文では Logcat を用いた動的解析による端末情報呼び出しの検知手法を考案した．この手法を用いることにより静的解析で検知することのできないような，難読化されたアプリケーションや，外部サーバと連携を行う機能を用いて攻撃用のコードを外部のサーバに設置しているアプリケーションの端末情報呼出しなども検知できる．また strace を用いた動的解析では検知できない API の端末情報呼出しも検知できた．今回の手法ではアプリケーションの挙動に着目しているため，あらかじめマルウェアのシグネチャを取得する必要がない．そのため本考案手法ならば未知のマルウェアも検知できる．

今後の課題としてマルウェアと正規のアプリケーションの区別することがあげられる．今回の手法はその特性上，API を通じて端末情報を呼び出したアプリケーションすべてを検知する．実際に利用するときはその中からマルウェアだけを抜き出して検知する工夫が必要となる．また，今回の研究では IPhoneSubInfo.aidl の中で定義されている API に関してのみ実験を行った．しかし，今回調査した以外の API に関しては実験できていない．今後の課題として，本考案手法が実際に他の aidl ファイルのなかで定義された API の検知にも利用できるのかを確かめる必要がある．

謝辞 本研究を進めるにあたり，有益な助言をいただいた株式会社 KDDI 研究所　窪田歩氏，同磯原隆将氏に感謝する．本研究の一部は，日本学術振興会 科学研究費補助金 基盤研究 C (21500078) による補助のもとで行われた．

参 考 文 献

1) 磯原隆将，川端秀明，竹森敬祐，窪田歩，可児潤也，上松晴信，西垣正勝：セカンドアプリ内包型 Android マルウェアの検知，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.708–713 (2011).
2) Permissions — Android Developers,
入手先⟨http://developer.android.com/guide/topics/security/security.html⟩
（参照 2012-02-14）.
3) Bornstein, Dan : Presentation of Dalvik VM Internals，入手先
⟨http://sites.google.com/site/io/
dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf⟩ Google. p. 22,（参照 2012-02-14）.
4) Yunhe Shi，David Gregg，and Andrew Beatty，M. Anton Ertl : Security and Virtual Machine Showdown: Stack Versus Registers,
入手先⟨http://www.usenix.org/events/vee05/full_papers/p153-yunhe.pdf⟩
Proceedings of the First International Conference on Virtual Execution Environments (VEE'05)，pp.153-163，USENIX，June 2005.（参照 2012-02-14）.
5) Android Interface Definition Language (AIDL) — Android Developers,
入手先⟨http://developer.android.com/guide/developing/tools/aidl.html⟩
（参照 2012-02-14）.
6) Ralf Benzmuller, Sabrina Berkenkopf : G Data Malware Report –Half-yearly report January - June 2011–，入手先⟨http://www.gdatasoftware.com/uploads/media/G_Data_MalwareReport_H1_2011_EN.pdf⟩（参照 2012-2-14）.
7) 川端秀明，磯原隆将，竹森敬祐，窪田歩：Android パーミッションを悪用する Script の脅威と静的解析，電子情報通信学会技術研究報告. ICM，情報通信マネジメント，111(30)，pp.13–18 (2011)，社団法人電子情報通信学会.