目次

1　はじめに

今日では，スマートフォンが非常に身近な存在であり，その中でも Android 端末は最も世界中で普及している． IDC による，世界中のスマートフォンの OS 別のシェアの調査 [1] においては，Android 端末は 80% 以上のシェアがあると示している． つまり，Android 端末は他の OS の端末，iOS, Windows Phone, Black Berry OS に比べて，より多くのユーザによって使われていることがわかる．Android がオープンソースであり，だれでも Android のソースコードを手に入れられることがその理由の 1 つである．そのため，様々なメーカーによって開発が行われ，価格，スペックに関して多種多様な製品が世界中で販売されている．

しかし，Android の普及に伴い，Android 端末を標的にした Android アプリのマルウェアによる被害が増えている．先に述べたように，Android はオープンソースであるため，攻撃者は脆弱性を見つけることは 他のモバイル端末 OS に比べると容易だ．Cisco の 2014 年のレポート [2] によると，モバイル向けマルウェアの 99.9% が Android を標的にしていると報告している．Y.Zhou, X.Jiang の研究 [3] では，彼らの研究に用いたデータセットの数は，2011 年の 6 月では 209 個だったのが，同年 10 月には 1260 個にも増加していた．2014 年の 9 月には，ロシアで銀行口座を狙った Android マルウェアを作成したとして，2 名の逮捕者が出ている．これらの事例を見てわかるように，Android 端末を標的にしたマルウェアによる被害は深刻であり，Android 端末のユーザは危険にさられている

これまで解析された多くの Android マルウェアが次の 2 つの挙動を示している．1 つは，個人，または端末の情報を抜き取ることである．感染した端末の情報を内部で収集し，この情報を外部へ送信する．例えば感染した端末に保存されている電話帳データを盗むことによって，マルウェア作成者は他の端末への攻撃が可能になる．もうひとつは，ユーザへの不正な課金である．バックグラウンドで SMS を送ることで，ユーザは不正に金額を請求させられてしまう．さらにマルウェアはユーザにメッセージを送信したことを気づかせないようにするので，ユーザは請求が来るまで全く気づかない．もちろん Android マルウェアはこれ以外にもさまざまな挙動を示す．このような被害を少なくするためには Android を標的にしたマルウェアを解析することはとても重要である．

code.google.com が提供している既存の Android マルウェアの解析ツールとして androguard[4] , droidbox[5] の 2 つがある．androguard は Android アプリのコード解析を行うことで，アプリ内のクラスごとの関係を示すグラフを作り，危険だと判定した部

分を赤く表示する．マルウェアが外部からコードをダウンロードして攻撃をする場合，静的解析では対応できないという問題点がある．droidbox はエミュレータ上でマルウェアを動かし，データのやりとり，ファイルの読み書き，などを動的に監視することでマルウェアの挙動を解析している．しかし，droidbox はマルウェアが実際に実機でどのような挙動をするかを正確にとらえているとは限らない．なぜならマルウェアがエミュレータ上で動いているのを検知して，挙動を変える可能性もあるからだ．

そこで本研究では実機における Android マルウェアの動的解析を提案する．マルウェアの実際の挙動をより詳細に調べるためには，実機でマルウェアを動かし，その挙動から解析を行う必要がある．マルウェアを実機で動かしながら，ログを得ることで解析を行う．提案手法をマルウェアに適用することで，実行されたメソッド名，クラス名，引数の型名と値を得ることができる．Android アプリは APK ファイルという 1 つのファイルにまとめられて端末にインストールされている．その APK ファイルから Java クラスファイルを取り出して，ログを得たいメソッドを含むクラスの Java クラスファイルを書き換える．Java クラスファイルを書き換えたマルウェアの APK ファイルを実機にインストールして動かすと，動的にログを得ることができる．このログを得ることでマルウェアのどのクラス，メソッドが不正な動きをしているのか，マルウェアがどのような情報にアクセスしているかがわかるため，解析を行うことができる．

本提案によりマルウェアを解析できたかを示すためにインターネットのサイトから入手した 11 個 のマルウェアを用いて 2 種類の実験を行った．1 つめの実験では，11 個の検体において，不正なコードを含むと思われるクラスのそれぞれのメソッドの始めにログを出力するようにクラスファイルを変更した．その結果，11 個中 5 個のマルウェアから，不正な挙動を表すログを得ることができた，具体的にいうと，SMS の送信や，外部からのコードの入手を示すログであった．2 つめの実験では，先の 11 個の検体の中の 1 つである，iMatch に対してのみ行った，1 つめの実験で行ったクラスファイルの変更に加えて，メソッド内でのメソッド呼び出しの情報も出力するようにした．この実験の結果として，iMatch の攻撃手段である，SMS 送信のための Android API とそのメソッドを呼び出しているメソッドとそのクラスを特定することができた．2 つの実験を通じて提案手法により一部のマルウェアの挙動を解析することができた．

本論文の構成を以下に示す．2 章では Android 端末を標的にした悪意あるマルウェアと Android アプリの基本的な構成について解説する．3 章では，Android アプリのマルウェアを解析している関連研究を紹介する．4 章では，マルウェアを解析するためにどのようにマルウェアの中にログコードを挿入するかについて説明する．6 章では，提案手法をマルウェアに適用させた実験とその結果について述べる．7 章では，まとめと今後の課

題について考察する.

2 Android を標的にしたマルウェア

本研究では，Android を標的にしたマルウェアの中で，悪意ある Android アプリを対象とする．2.1 では，悪意あるアプリの挙動，不正な振る舞いをするためにどのような方法をとっているのかについて説明する．また，マルウェアの例の 1 つとして，GoldDream の挙動を説明する．2.2 では，基本的な Android アプリの構成について説明する．Android アプリの概要を示している AndroidManifest.xml とアプリの実行ファイルである，classes.dex について説明する．

2.1 悪意ある Android アプリ

マルウェアの主な挙動として，個人情報の盗難と不正な金銭請求がある．個人情報に関して言えば，デスクトップ PC やノートパソコンに比べ，スマートフォンは，電話帳，メールなど個人情報のデータの量が多いため，攻撃者の標的になりやすいのは明らかである．スマートフォンでもネットサービス等で銀行口座の操作ができるため，スマートフォンのブラウザに銀行アカウントのパスワードが残っている可能性もある．もし銀行アカウントのパスワードが盗まれた場合，多額の被害を生んでしまうおそれがある．ユーザ自身の情報だけでなく，端末の情報，IMEI（端末識別番号），SIM カードの情報, GPS の位置情報なども盗まれている．金銭を不正に請求するための攻撃方法として，SMS (Short Message Service) を使ったものがある．SMS を使った攻撃の様子を 図 2.1 に示す．SMS Premium Service は，ある番号へ SMS を送ることで音楽や動画などのコンテンツを買うことができるサービスである．この攻撃は Premium Service のように，マルウェアが攻撃者たちの番号へ SMS を送信することで，ユーザーに課金させる方法だ，その課金は携帯電話の料金の支払いと同時に行われ，そこで支払われた料金の一部が攻撃者たちに支払われる．ユーザはその支払い請求が来るまで SMS が送られたことに気づかない．通常の Premium Service ではユーザに支払い確認のメッセージがくるのだが，マルウェアはこれをブロックするためだ．

マルウェアが先に述べたような攻撃をするための方法を 2 つ挙げる． 1 つは，外部からの遠隔操作だ [6] . マルウェアは外部サーバからの命令を受け取り，実行する．あるマルウェアがインストールされると，外部のサーバから暗号化されたスクリプトを受け取り，その復号，実行するという例もある．この手法を使うと，マルウェアを検知するソフトウェアを回避することもできる．なぜなら，公式アプリストア (Google Play) にアッ

プロードされた時点では不正な動きをするコードをマルウェア自身は何も持っていないため，検知されないからだ．外部から得たスクリプトは DexClassLoader というクラスローダによりアプリケーションに組み込まれていないファイルを読み込むことができる．もう一つは，Android OS の脆弱性をつくことで，特権レベルを上げることである．不正にマルウェア自身の特権レベルを上げるマルウェアの中には root 権限を奪うものもある．マルウェアに root 権限を奪われてしまうと，ユーザが抵抗できる余地は少ないため，悪用されると非常に危険である．マルウェアの 1 つである，AndroidDefender [7] は，表向きにはウイルス対策アプリとなっている．AndroidDefender が起動すると，それは感染した端末から電話をかけられなくしたり，さまざまなアプリケーションへのアクセスを制限させる．その後，AndroidDefender は端末を修復するためにユーザに大金を要求する．

実際のマルウェアの例として，GoldDream を紹介する．GoldDream は端末情報を流出させるために，電話の発着信などのイベントを監視し，外部へと送信する． そこで，GoldDream の挙動について説明する．GoldDream は SMS の受信，電話の発着信があると，バックグラウンドでユーザに気づかれることなく起動される．GoldDream はレシーバを登録することで，これらの着信が来た時に Android OS が出す通知を受け取れるようにしている．SMS を受信した際は，受信したメッセージの送り元のアドレス，内容，タイムスタンプを収集する．電話の発着信の場合も同様に，電話番号やタイムスタンプといった情報が GoldDream によって集められる．なぜこのような情報をマルウェア作成者が盗もうとするかというと，この情報を用いることで，他の端末へ攻撃を拡大できるためである．例えば，他の端末の情報を得ることができれば，感染した端末から他端末へマルウェアのダウンロードリンクを載せたメッセージなどを送ることができる．これらの情報は一度ローカルファイルに保存された後，外部のサーバへ送信される．GoldDream は外部サーバからコマンドを受け取り，実行する．サーバから受け取るコマンドは，次の４つである．１）SMS をバックグラウンドで送信する，２）電話を発信する，３）アプリをインストール，アンインストールする，４）ファイルをサーバへアップロードする．ファイルをアップロードするコマンドは端末の情報を送信するために用いられる．

2.2 Android アプリの構成

1 つの Android アプリは 1 つの APK ファイル (.apk) となってまとめられている．Android のアプリを実行するためには，異なる種類の複数のファイルが必要である．例えば，AndroidManifest.xml, 画像，レイアウトファイル (png, jpg, xml, etc)，classes.dex, アプリの証明書，である．これらを 1 つのファイルに ZIP 形式でまとめたものが APK

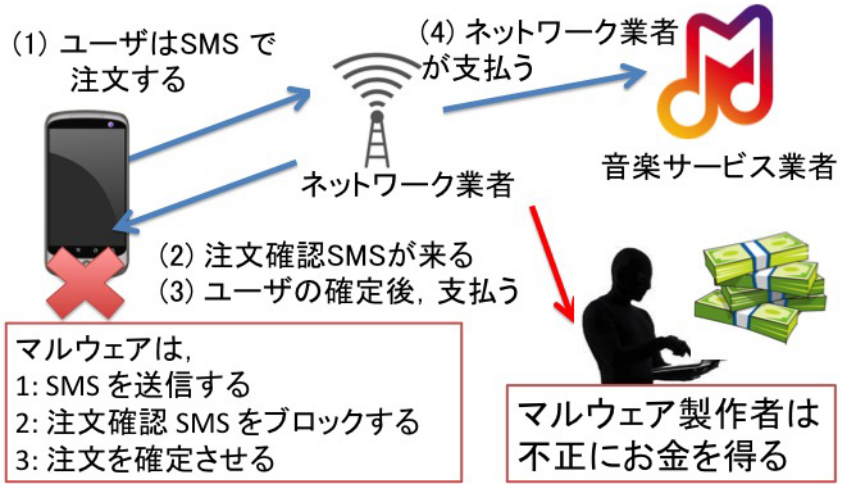


図 2.1　SMS を使った攻撃

ファイルである．そのため，zip ファイルと同様に解凍，圧縮，中身の入れ替えができる．そのため，.zip ファイルを解凍するのと同様, unzip コマンド で APK ファイルを解凍することができる．APK ファイルを解凍すると，AndroidManifest.xml, classes.dex, res ディレクトリ，META-INF ディレクトリが展開される．res ディレクトリには，アプリのアイコン画像, アプリ実行時に画面上に表示する画像ファイルが入っている．META-INF ディレクトリにはアプリの証明書のファイルがある．ただし，unzip コマンドで解凍した場合，AndroidManifest.xml はバイナリのままであるため，このファイルの中身を見たい場合は，apktool [8] というツールを使う必要がある．なぜこれらのファイルを一つの APK ファイルにまとめないといけないかというと，他のアプリも同じファイル名を用いているためだ．どのアプリも必ず AndroidManifest.xml と classes.dex の 2 つのファイルを持っている．そのため，これらのファイルはアプリごとにまとめて端末上にインストールされる必要性がある． そうすることで，Android OS はアプリケーションを管理することができる．

AndroidManifest.xml とはアプリの基本的な情報が書かれている XML 形式のファイルである．あるゲームアプリの AndroidManifst.xml の例を図 2.3 に示す．アプリのパッケージ名，アプリが使用する権限，アプリが起動した時に最初に実行されるクラス，などが記されている． パッケージ名は OS がアプリを識別する名前である．図 2.3 の 2 行目に．このアプリのパッケージ名 (com.gamelio.DrawSlasher) が書いてある．待ち受け画面で，アイコンの下に表示される名前とは異なる．例えば，Facebook，Instagram の Android アプリの場合はそれぞれ， com.facebook.katana，com.instagram.android

となっている．一般的にアプリを使用している分には，ユーザはパッケージ名について気にする必要がないので，使用していてアプリのパッケージ名を目にすることはまずない．ただし，adb (Android Debug Bridge) を用いてターミナルからアプリを手動でアンインストールする場合は，パッケージ名を特定する必要がある．OS monitor という Android のシステム状況を確認できるアプリを使うと，AndroidManifest.xml を見なくても，実行中のアプリのパッケージ名を端末上で見ることができる．Android のアプリは OS から権限を得ないと実行できないことがいくつもある．電話の着発信，SMS の送受信， インターネットへの接続などである．AndroidManifest.xml に記すことにより，アプリはその権限を得る．これらの機能をアプリで実行するためには，必ず AndroidManifest.xml に宣言しないといけない．図 2.3 の中では，24 行目から 38 行目にかけてこのアプリの権限が書かれている．さらに，マルウェアの AndroidManifest.xml を得ることができれば，どのようなことをしようとしているのかがわかる．表向きは電話帳のデータとは関係の無いアプリであるのに，AndroidManifest.xml で電話帳へのアクセスの権限を要求していたら，何らかの不正な動きをするアプリである可能性であることが高い． 図 2.3 の 34 行目には，電話をかけるための権限である，CALL_PHONE が宣言されている．このアプリがマルウェアではない，安全なゲームアプリであるとすれば，電話を発信する必要はない．つまり，このアプリは電話をかけることで何らかの不正な動きをする可能性が高い．また，AndroidManifest.xml ではアプリが起動したときに最初に実行する activity を指定する必要がある．図 2.3 の 7 行目から，最初に実行される activity は com.Claw.Android.ClawActivity ということがわかる．この指定が無いと Android OS はどこから実行すればよいかわからない．activity とは 1 画面を表すクラスであり，Android アプリ内で画面が変わるということは，他の activity に変わる（遷移する）ということである．画面内でボタンなどを表示させたいときは，android.Activity クラスをオーバーライドして，実装する．このように，AndroidManifest.xml はアプリの大まかな概要を示している．

アプリの画像ファイルが入っている res (resource) ディレクトリ内には様々なファイルがあり，その種類に応じて配置されるディレクトリが決まっている [9] ．図 2.2 は，あるアプリの res ディレクトリの構成を示したものである．layout ディレクトリでは，ユーザインターフェース (以下 UI) のレイアウトを定義する XML ファイルが入っている．レイアウトリソースファイルはアプリの activity の UI の構成を決めている．例えば，テキストを表示する領域を表す TextView の縦幅と横幅の具体的な値が記されている．drawable ディレクトリには画像ファイル (.png, .jpg, gif etc) と形状などを定義した XML ファイルがある．drawable ディレクトリにある XML ファイルはボタンを押した

時などの状態変化のために用いる．図 2.2 では，drawable ディレクトリがいくつも分かれている．これは，ことなる解像度に対応するためである．ldpi は Low-density,，mdpi は Medium-density，xhdpi は Extra-high-density, xxhdpi は Extra-extra-high-density を表す．menu ディレクトリにはアプリのメニューの内容を定義する XML ファイルが．values ディレクトリには文字列，数値，色などの値を定義する XML ファイルが入っている．

classes.dex は，Android アプリの DEX コード実行ファイルである．DEX コードとは，Android 上で動く VM，Dalvik VM の中間言語だ．Android アプリの APK ファイル内には，必ず classes.dex は 1 つしか存在しないことになっている．二つ以上の .dex ファイル（例えば，classes.dex と classes1.dex）を APK ファイル内に入れることはできるが，実行するときは classes.dex のみが実行される．アプリのソースコードの全てのクラスファイルの中身が 1 つの DEX コードに変換される．Java VM も中間言語である，Java バイトコードを用いている．Java では，コンパイル時にクラス毎に Java バイトコードのファイル，クラスファイルが生成される．しかし，Dalvik VM では，アプリごとに DEX コードのファイル (classes.dex) が生成される．Android アプリを実装していた場合，自分が書いた Java ファイル (.java) がコンパイルされて Java クラスファイル (.class) に変換され，そのクラスファイルが 1 つの DEX コードファイルにまとめられるという流れになる．つまり， Java で実装されたソースコード中のクラスとは関係なく，1 つのファイルになる．また，dex2jar [10] というツールにより，classes.dex を JAR 形式に変換することができる．JAR ファイルは Java バイトコードが圧縮されたファイルであるから，これを解凍することで，Android アプリのクラスファイルを手に入れることができる．また，Android SDK が提供する dx というコマンドラインツールを用いることで，jar ファイルから DEX コードファイルを作成することもできる．本提案ではこれらの方法を用いてマルウェアのクラスファイルを入手し，そこから classes.dex を作成した．

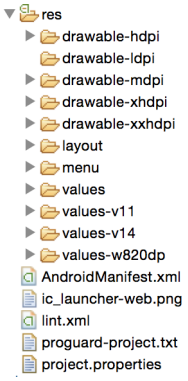


図 2.2　res ディレクトリの構成

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1" android:versionName="1.0.1" android:installLocation="preferExternal" package="com.gamelio.DrawSlasher"
  xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-permission android:name="android.permission.WAKE_LOCK" />
    <uses-permission android:name="android.permission.INTERNET" />
    <application android:label="Blood vs Zombie" android:icon="@drawable/icon">
        <activity android:name="com.Claw.Android.ClawActivity" android:screenOrientation="landscape" android:configChanges="keyboard|keyboardHidden|orientation">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name="com.GoldDream.zj.zjReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
                <action android:name="android.provider.Telephony.SMS_RECEIVED" />
                <action android:name="android.intent.action.PHONE_STATE" />
                <action android:name="android.intent.action.NEW_OUTGOING_CALL" />
            </intent-filter>
        </receiver>
        <service android:label="Market" android:name="com.GoldDream.zj.zjService" android:exported="true" android:process="" />
    </application>
    <supports-screens android:smallScreens="false" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    <uses-permission android:name="android.permission.SEND_SMS" />
    <uses-permission android:name="android.permission.READ_SMS" />
    <uses-permission android:name="android.permission.CALL_PHONE" />
    <uses-permission android:name="android.permission.PROCESS_OUTGOING_CALLS" />
    <uses-permission android:name="android.permission.DELETE_PACKAGES" />
    <uses-permission android:name="android.permission.INSTALL_PACKAGES" />
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
</manifest>
```

図 2.3　AndroidManifest.xml の例

3 関連研究

3.1 では 1 章で述べた，androguard, droidbox についてより詳しく説明する．3.2 で，Android マルウェアを解析，調査した研究を 3 つ紹介する．

3.1 Android マルウェアを解析するためのツール

androgurard [4] は Android マルウェアを解析し，クラス同士の関係を表すグラフを作る．この Android アプリのクラスの視覚化ツールは Androgexf である．androguard は Androgexf だけではなく, 他にもたくさんのツールを提供しており, その数は全部で 14 にもなる．例えば，その中の 1 つの Androaxml は 2.2 で説明した，AndroidManifest.xml などのバイナリーの XML を人間が読めるように変換するツールである．Androgexf に APK ファイルを入力すると，GEXF 形式のファイル (.gexf) として解析結果のグラフを出力する．GEXF ファイルは Gephi というフリーソフトで見ることができる．このグラフはメソッドコールグラフであり，それぞれのノードには，メソッドのタイプ (activity, service, receiver etc) , クラス名，どの権限が使用されているか，その権限のレベル (それぞれの権限には，normal, signature, dangerous などのレベルがある) が記されている．図 3.1 のように，このグラフではどの部分が危険であるかを色を変えて示してあり，どのメソッドが不正な動きをしているかがグラフを見れば簡単にわかる．図 3.1 では，”MONEY_RISK”, ”SMS_RISK” というタグが付けられている．これは，矢印が指されているメソッドのノード，sendSms がバックグラウンドで SMS を送るという危険があることを示している． また，動的にコードをロードしているメソッドも検知することができる．

droidbox [5] は Android のエミュレータ上でマルウェアを実行することで動的に解析する．droidbox は，端末のネットワークデータのやりとり，ファイルの読み込み・書き込みの命令，ネットワーク，ファイル，または SMS を通じた情報の流出，送信された SMS と電話，開始されたサービス，といった多くの情報を解析する．さらに，解析後は，マルウェアの振る舞いを表す 2 つのグラフも生成する．図 3.2 , 図 3.3 はこの 2 つのグラフである．図 3.2 の縦軸はアプリの活動の種類を表し，横軸は時間である．図 3.2 の 横軸 30 から 40 にかけて，”net open”, ”leak” という activity が頻繁に発生している．これはこの間に情報が流出したことを意味する．図 3.3 は解析した複数のマルウェアの類似性を比較するために用いられる．それぞれの色 (CALL, FILE WRITE, NETLEAK,

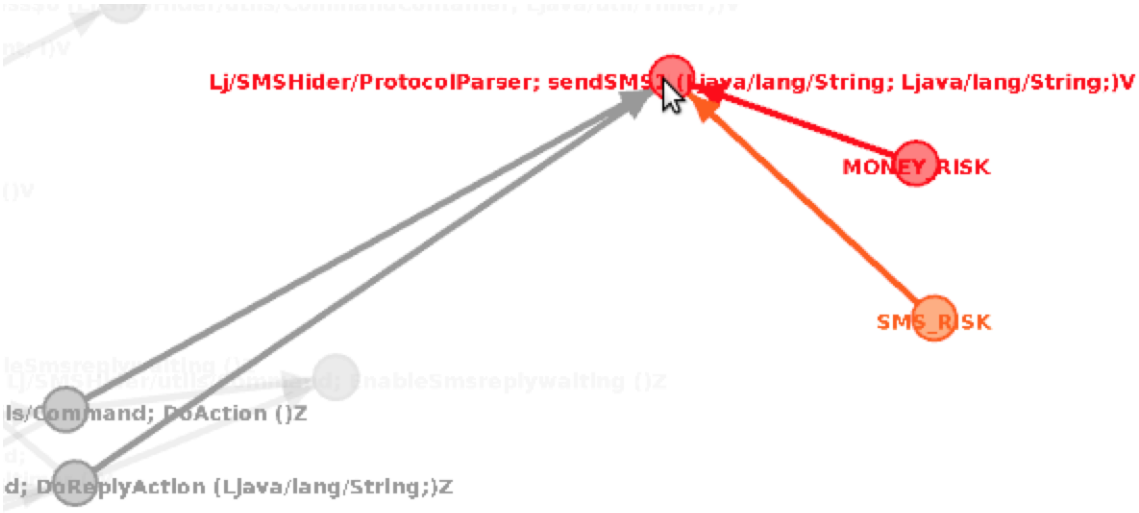


図 3.1　androguard が生成するグラフの一部

etc) の領域の割合を比べることで，異なるパッケージのマルウェアがどれほど似た挙動をしているかがわかる．

3.2　Android マルウェアを解析，調査した研究

Y.Zhou, X.Jiang は，2010 年の 8 月から 2011 年の 10 月にかけて，公式サイト，非公式サイトから収集した 1,260 個，49 種類の Android マルウェアを用いて時系列調査と分類調査を行っている [3] ．時系列調査では，この研究で収集しているマルウェアの数 (dataset) をグラフで示している．図 3.4 は その dataset の推移を表すグラフである．DroidKungFu が登場した 2011 年 6 月，AnserverBot が登場した 2011 年 10 月にこの dataset の数が急激に増えた．つまり，この 2 つのマルウェアが大きな影響を及ぼしていることがわかる．分類調査では，マルウェアのインストールの方法，起動トリガー，挙動，マルウェアが要求する権限を調査している．49 種類中，25 種類のマルウェアが Repackage によりインストールされていた．マルウェアの起動トリガーとして最も多かったのは，OS の起動時で，29 種類だった．マルウェアの挙動は，Financial charge が最も多く，その中でも，SMS を使っているものが多かった．また，多くのマルウェアが SMS, Wi-Fi に関する権限を要求していた．また，先に出てきた，2 つのマルウェアがどのような挙動をするかを示している．DroidKungFu は 6 種類のヴァージョンが見つ

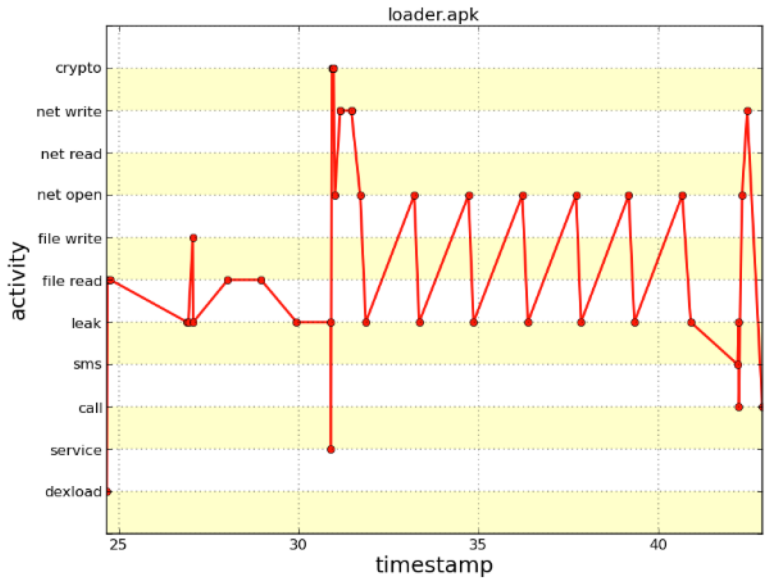


図 3.2　droidbox の activity の時系列グラフ

かっており，外部サーバのアドレスの格納方法が暗号化によって複雑化している．そのため，全く意味の無い文字列でも暗号化されている可能性があるのでこれらを復号して確認する必要が出てくる．よって，解析により手間がかかり，さらに難しくなってしまう．AnserverBot は解析回避と遠隔操作の 2 つの特徴を持つ．AnserverBot は解析回避のために 2 つの方法をとっている．一つは自身が解析されているかどうかを検知する方法で，もう一つはメソッド名やクラス名を意図的に変えることで解析を行いにくくする方法である．

さらに，彼らは既存の 4 つのセキュリティソフトがこの dataset を検知するかどうかの調査も行った．その結果，最高は Lookout の 79.6 % , 最低は Norton の 20.2 % で，どのソフトウェアも検知できないマルウェアも存在した．この結果より，この 4 つのセキュリティソフトはまだ不十分であることがわかる．これらの調査結果からこの研究の結論として，マルウェアのインストール方法の中でも，最も頻繁に行われている Repackage を検知することとアプリの外部のコードの動的ローディングを防ぐ技術が必要であると彼らは主張している．この研究は Android マルウェアを調査，分類し，さらに 2 つのマルウェアについて詳しく挙動を示している．この研究では，解析手段（静的か動的か）までは言及していない．本研究では調査，分類を行うために必要となるさまざまな種類のマルウェアの挙動を正確に解析する手法を提案する．

S.Poeplau らは，マルウェアの動的に外部コードの動的ローディングに焦点をおいて

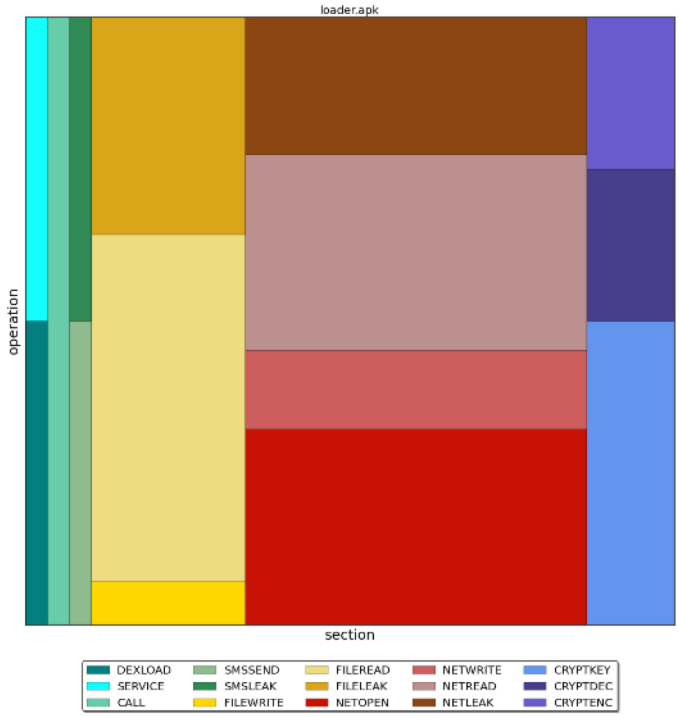
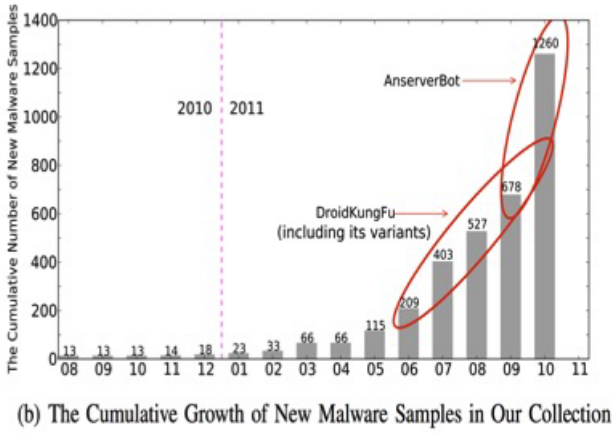


図 3.3　droidbox の operation - section グラフ

解析を行っている [11]．2.1 で述べたように，外部コードのローディングをすることで公式ストアの検知システムをくぐり抜けることができる．また，外部コードのローディングは必ずしも不正なものではなく，マルウェア以外のアプリでも使われている．しかし，Android OS はロードされたコードをチェックしないので攻撃者はロードするものを置き換えることができる．そのためこれは Android アプリの脆弱性といえる．そこで，この研究で彼らは外部コードの動的ローディングを検知するツールを提案している．このツールは APK ファイルから取り出した DEX コードを静的に解析する．100 万回以上インストールされた，1,632 個のアプリをランダムに選び，このツールを用いて検査した．その結果，その中の 9.25 % から外部コードのローディングの脆弱性が検知された．さらに，Google Play での人気 50 位以内の無料アプリを同じツールで検査すると，16 % ものアプリがその脆弱性を示した．また，彼らはこの攻撃手法に対する防御策も提案している．Dalvik VM が外部からダウンロードされたコードのハッシュ値を計算し，それが Whitelist に載っていなければ，それを実行できないようにアプリケーションに制限をかける．そうすることで，これを利用した攻撃を防ぐことができる．外部コードの動的ローディングの応用例として，文字列としてクラス名，メソッド名を受け取り，Java の

reflection を通して実行することが考えられる．reflection とは，プログラムの実行時においてプログラム自身の構造を読み取ったり書き換えたりする技術のことである．彼らの研究では外部コードの動的ローディングの攻撃を防ぐツールを提案したが，reflection を使うと，このツールでは検知することができない．この研究では静的解析を行っているために，外部コードのローディングは検知するが，外部コードの挙動が悪意あるものかどうかは解析しない．上記でも述べたように，外部コードのローディングは必ずしも悪意あるものとは限らない．それに対して本研究では，動的解析を行っているためにこのような外部コードの動的ローディングによる悪意ある挙動を解析することができる．

L.Yan, H.Yin はマルウェアの動的解析環境 (DroidScope) を提案している [12] ．彼らは Android SDK が提供するエミュレータをベースに自分たちで手を加えて，そのエミュレータの中でマルウェアを動かしている．彼らは Android システム全体の再構築を行っている．つまり，DroidScope はハードウェア，Linux OS (Android は Linux をベースにして動作している)，Dalvik VM の 3 種類の API を提供する．DroidScope が提供する API を用いることで，Android API, Dalvik VM, Linux , さらには機械語の命令までをトレースするツールを提案している．*API tracer, native tracer, Dalvik instruction tracer* の 3 つである．また，これらの API に動的な taint analysis を実行することで，情報漏えいを解析するツール (*Taint tracker*) も提案している．彼らはこの 4 つのツールのパフォーマンス測定も行っている．元のエミュレータで実行時間を基準に，4 つのツールの実行時間を調べた．その結果，オーバーヘッドは小さいと言える結果であった．しかし，taint traker は他の 3 つのツールとくらべて大きなオーバーヘッドを示した．彼らはこれらのツールを用いて先に述べた，DroidKungFu を解析した．この解析によって，このマルウェアのルート権限を取得する方法と情報を盗み出す方法を明らかにしている．DroidKungFu だけでなく，論文中では DroidDream も解析している．DroidKungFu の場合と同様な解析を行った結果，DroidDream が端末識別番号を盗み出す方法を突き止めた．DroidScope は 1 つの実行パスのみを解析しない．彼らは実行パスを増やすために，システムコール，ネイティブ API，Dalvik メソッドなどの返り値を変えることで，異なる実行パスを実現した. symbolic execution のほうが，より良いことが考えられるが，かれらはこれを今後の課題としている．彼らの研究は実行された API，メソッドを動的に得ることで解析を行っているため，解析のためのアプローチは本研究と共通している点がある．しかし，本研究は実機で行っているのに対して，彼らの研究はエミュレータ上で行っている．もし，マルウェアがエミュレータ上で動作しているのを検知して，振る舞いを変える可能性もある．デスクトップ PC を標的にしたマルウェアは実際に振るまいを変えることがわかっている [13] [14] [15] . DroidScope のような解析ツールが普及するに



(b) The Cumulative Growth of New Malware Samples in Our Collection

図 3.4　マルウェアの dataset の推移

つれて，エミュレータでの実行を検知するマルウェアが出てくるだろうと，彼らはこの論文中で述べている．本研究が提案している実機による解析であればマルウェアの ”正常” な動作を解析することができる．

マルウェアを解析している方法としては，動的解析と静的解析がある．S.Poelau らは外部コードの動的ローディングを用いるマルウェアを静的解析により検知する手法を提案している．しかし，静的解析であるために，実際に外部からコードを受け取ってその結果実行されるマルウェアの挙動を解析することはできない．L.Yan, H.Yin が提案する DroidScope はエミュレータ上で動的にマルウェアを解析している．エミュレータで実行する場合と実機で実行する場合では，厳密に同じ環境であるとはいえない．エミュレータでの解析を検知する機能を持ったマルウェアが将来出てきた場合，DroidScope はマルウェアを解析することができなくなってしまう．そこで，本研究では実機でマルウェアを実行して解析を行う．

4 提案

本研究では，Android を標的にしたマルウェアの動的解析を行う．マルウェアにログコードを挿入させ，そのログを動的に得ることで解析を行う．エミュレータでは再現できないために解析できないことも，本提案を適用することで，実機でマルウェアを動かすため解析できるというメリットがある．4.1 では，本提案の全体の流れを説明する．4.2 では，本提案の解析手順を，4.3 では，ログコードをどこに挿入するかについて説明する．

4.1 概要

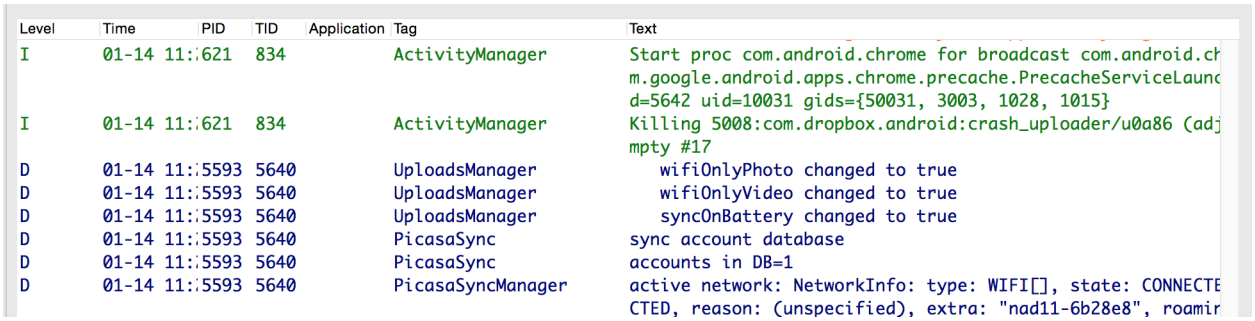
本研究で対象とする Android アプリのマルウェアを解析するためにはメソッドの情報が必要である．解析を行っていくためには，マルウェアがどのようなコードが実行したかということを知る必要がある．Android アプリのマルウェアは Java で実装されているため，解析の手がかりとなるのはメソッドのログである．このログにはマルウェアがどんなメソッドを実行したかという情報とそれぞれのメソッドについての情報が含まれている．メソッドの情報とは具体的にいうと，メソッド名，そのクラス名，引数の型名と値である．本提案では，このログを用いてマルウェアの解析を行う．

本提案では，先に述べたログを得るためにマルウェアのメソッドにログを出力させるコードを挿入する．Android API では，デバッグ用等のために，ターミナルにログを出力する API を提供している．その API とは android.util.Log である．以下に示す例のようなコードをメソッドの中に挿入する．Log.d の ”.d” は ”DEBUG” を表しており，Android のログの重要度を示している．ログコードを挿入してマルウェアを実行させると，マルウェアがそのメソッドを実行したときにそのログコードは必ず実行される．よってメソッドについてのログを出力させることができる． 4.2 では，どのようにコードを挿入するかの手順について説明する．

android.util.Log の例

```
Log.d(”Tag”, ”Log Message”);
```

本提案では，まずマルウェアの APK ファイルを入手することから始まる．次に，APK ファイルから Java クラスファイルを取り出し，ログをターミナルに出力させるためのコードを挿入する．このログには，実行したメソッド名，そのメソッドの引数の型名とその値を出力させるようにする．コードを挿入するとは，マルウェアから取り出した



Level	Time	PID	TID	Application	Tag	Text
I	01-14 11::	621	834		ActivityManager	Start proc com.android.chrome for broadcast com.android.ch m.google.android.apps.chrome.precache.PrecacheServiceLaunc d=5642 uid=10031 gids={50031, 3003, 1028, 1015}
I	01-14 11::	621	834		ActivityManager	Killing 5008:com.dropbox.android:crash_uploader/u0a86 (adj mpty #17
D	01-14 11::	5593	5640		UploadsManager	wifiOnlyPhoto changed to true
D	01-14 11::	5593	5640		UploadsManager	wifiOnlyVideo changed to true
D	01-14 11::	5593	5640		UploadsManager	syncOnBattery changed to true
D	01-14 11::	5593	5640		PicasaSync	sync account database
D	01-14 11::	5593	5640		PicasaSync	accounts in DB=1
D	01-14 11::	5593	5640		PicasaSyncManager	active network: NetworkInfo: type: WIFI[], state: CONNECTE CTED, reason: (unspecified), extra: "nad11-6b28e8", roamir

図 4.1　Android が出力するログの例

Java クラスファイルを書き換えるということである．マルウェアにコードを挿入した後，書き換えた Java クラスファイルを DEX コードに変換し，classes.dex を作成する．classes.dex を作ったら，元の APK ファイルの中にあるオリジナルの classes.dex とコードが挿入されている新しい classes.dex を入れ替える．ログコードを挿入したマルウェアを Android 端末にインストールした後，DDMS (Dalvik Debug Monitor Server) というツールを用いることで，Android OS が出す多数のログの中からマルウェアが出したログを抜き取る．図 4.1 は Android OS が出力するログの例である．先に示した android.util.Log の例で示したように，それぞれのログにはタグがある．例を出すと，図 4.1 の 3 つめのログのタグは ”UploadsManager” であることがわかる．さらに，一番左の項目は ”D” となっていることから，このログの重要度は ”DEBUG” レベルであることがわかる．Android のログのタグは自分自身で決めることができるので，このタグを使って，本提案によるマルウェアのログのみを得ることができる．マルウェアのログを得ることで，マルウェアが実行したメソッドとその情報がわかるので，そこからマルウェアの挙動を解析できる．

4.2　解析手順

最初に，大まかな手順の流れを説明する．Step 1 で，APK ファイルから Java クラスファイルを取り出す． Step 2 では，その Java クラスファイルにログコードを挿入する．Step 3, 4 でそのログコードを挿入した APK ファイルを作成し，実機にインストールする．その後，マルウェアを起動して，ログを得る (Step 5, 6)．以下にそれぞれの Step の詳細な説明をする．

[Step1] まず最初の手順として，APK ファイルから Java クラスファイルを取り出す．2.2 で述べたように，APK ファイルを解凍することで，classes.dex を得ることができる．

classes.dex から Java クラスファイルを取り出すために，dex2jar [10] が提供する sh プログラム (d2j-dex2jar.sh) を用いる．これにより， DEX コードファイルを JAR ファイルに変換することができる．よって，APK ファイルを解凍して出てきた classes.dex から dex2jar を用いて 1 つの JAR ファイルを得ることができる．この JAR ファイルを解凍することで Android アプリの Java クラスファイルを得る．

[Step 2] マルウェアの Java クラスファイルを得た後は，これにログを出力するコードを挿入する．つまりマルウェアの Java クラスファイルを書き換える．本研究では，Java クラスファイルを書き換えるために Javassist [16] という Java ライブラリを用いる．Javassist は Java バイトコードの知識があまりなくても バイトコード変換のための API を提供する Java ライブラリである．Java で実装したプログラムでマルウェアの Java クラスファイルを書き換える．実装したプログラムについては，5 章で詳しく説明する．

[Step 3] Java クラスファイル を書き換えた後は，classes.dex を作成する．classes.dex を作成するためには，複数のクラスファイルを JAR ファイルにまとめる必要がある．dex2jar で得た JAR ファイルを解凍した際に，ディレクトリがいくつも出てくる場合がある．その場合は，ディレクトリ毎に JAR ファイルにまとめ，ディレクトリに属していないクラスファイルだけで，ひとつの JAR ファイルにまとめる． そして，以下に示すコマンドで JAR ファイルから classes.dex を作成する．dx とは，Android SDK が提供する dx コマンドのことである．このコマンドにより，JAR ファイルから DEX コードの変換を行う．

JAR ファイルから classes.dex を作る dx コマンド

```
dx –dex –output=”classes.dex” ”direcA.jar” ”direcB.jar”
```

[Step 4] 次に，新しく作成した classes.dex を APK ファイル内にあるオリジナルの classes.dex と入れ替え．端末にインストールできるように APK ファイルにサインを行う．先に述べたように，APK ファイルは ZIP 形式であるから， zip コマンドに -u オプションをつけることで，新しいファイルを古いものと入れ替えることができる．APK ファイルにサインするためには，dex2jar の中の d2j-apk-sign.sh を用いる．例えば，sampleApp.apk に対してこのプログラムを実行すると，sampleApp_signed.apk のように別の新しい APK ファイルが生成される．そして，adb shell を使って，この APK ファイルを実機にインストールする．

[Step 5] マルウェアのインストール後，手動でそのマルウェアを起動し，DDMS に出力されるマルウェアのログをテキストファイルに保存する．Android OS が出力しているログは，ターミナルでも見ることができるが，細かい内部システムの状態などの情報が大

量に出てくる．そのため，マルウェアが出しているログをそこから見つけることは困難である．DDMS では，指定したタグのみを出力することができる．マルウェアに挿入するコードには，タグを自分で指定しているため，DDMS にはマルウェアの実行ログのみを表示することができる．そして，表示されているログをテキストファイルとして保存する．

[Step 6] さらに，ログをより解析しやすくするために，このマルウェアのログのテキストファイルをクラス毎に分割する．なぜなら，そのままの状態では，複数のクラスのメソッドのログが混在していて，何が行われているか理解しにくいためだ．ログをパースするスクリプトを実装し，それを得られたテキストファイルに適用し，クラス毎に分割した．このスクリプトについては， 5 章で詳しく説明する．

4.3 ログコードの挿入箇所

マルウェアの挙動を解析するためには，ログコードを適切な箇所へ挿入する必要がある．そこで本節では，マルウェアの Java クラスファイルへログコードを挿入する際に，どのような箇所へ挿入するか，どのような情報をログコードで出力させるかについて説明する．4.3.1 では，メソッドの先頭へのコード挿入について，4.3.2 では，メソッド呼び出しの前後でのコード挿入について，なぜそこへ挿入するかという理由も含めて説明する．

4.3.1 メソッドの先頭へのコードの挿入

メソッドの先頭へログコードを挿入するのは，メソッドが実行された時に，そのログコードを確実に実行しそのログを出力させるためである．Javassist が提供する API では，メソッドの先頭か，最後にコードを挿入できる．メソッドの最後にコードを挿入してしまうと，メソッドが実行されたときに，そのコードが確実に実行されるかどうかは分からない．理由は主に 2 つ考えられる．1 つは，メソッドの途中で return 文が書かれている場合である．if 文の中で return 文が書かれていて，ある条件ではその if 文が実行されて，メソッドの最後まで到達せずに呼び出し元へ返ってしまい，メソッドの最後にあるログコードは実行されなくなってしまう．また，マルウェア作成者が解析者の混乱を誘うために，意図的にメソッドの途中で return 文を置いている可能性もある．もう一つの理由は，そのメソッドが途中で他のメソッドを呼び出してそのメソッドの最後まで到達しない場合だ．例えば メソッド A，B の最後にログコードを挿入し，メソッド A の中でメソッド B を呼びだすというケースを考える．そして，メソッド B の中でそのアプリが終了する関数が最後に呼ばれたとする．そうすると，メソッド A だけでなく，メソッド B

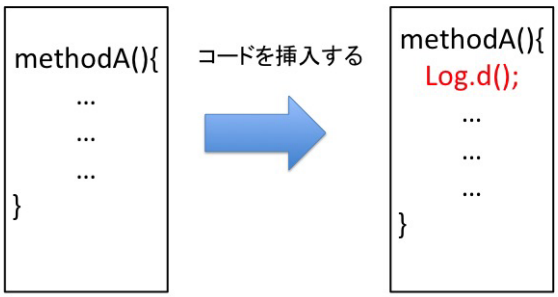


図 4.2　メソッドの先頭へコードを挿入するイメージ図

のログコードも実行されなくなってしまう．なぜなら，メソッド B のログコードはアプリを終了させる関数の後に挿入されることから，このログコードは実行されないからだ．もし実行フローが最後まで到達して，ログコードが実行されたとしても，解析する際には時系列とは逆の順番でログが出力されるので紛らわしくなってしまうというデメリットもある．メソッドの先頭へログコードを挿入するイメージ図 5.1 を以下に示す．図 5.1 が示すように各メソッドのメソッド先頭，つまりそのメソッドが実行された場合，この挿入されたコードが一番最初に実行されるということになる．よって，メソッドの先頭にログコードを挿入すると，メソッド内の条件等にかかわらず，必ずそのログコードが実行されることになる．

メソッドの先頭に挿入するログコードではクラス名，メソッド名，そのメソッドの引数の情報を出力させる．メソッドの引数の情報とは，引数の型名とその引数の値である．なぜ引数の型名だけでなく，中身を出力させるかというと，引数の中身がわかるとマルウェアの挙動がより見えやすくなるからである．例えば，メソッド名が ”getCode”，メソッドの引数の 1 つの型が String 型とわかっていたとする．しかし，これだけでは，String の中身がどんなものなのかよくわからない．なぜなら，String 型の変数はいろんな使われ方が考えられるからだ．型名だけで類推することはできても特定するのはとても困難である．この引数はサーバからのコマンドや，盗んだ端末についての情報の文字列かもしれない．そこで，引数の具体的な値が，URL のような文字列であることがわかると，この引数はコードを取ってくるアドレスということである確率がとても高いといえる．同様なことが int 型の場合も考えられる．もし int 型のメソッドの引数が携帯電話の番号（11 ケタで，090 や 080 で始まっている数字列）だった場合，そのマルウェアは感染した端末の電話番号を盗んでいたり，SMS をバックグラウンドで送信している可能性が出てくる．たとえメソッド名がマルウェア作成者により意図的に変えられていたとしても電話番号を

引数にとっているということは端末の情報への不正なアクセスを明らかに示している．このようにメソッドの引数の値はマルウェアを解析するにあたって重要な要素であり，この情報によって解析がより行いやすくなる．

しかしこの手法を適用できないメソッドが 2 種類ある．1 つは java.lang に属するクラスである．たとえば，String クラスのメソッドである，toString() の先頭にはログコードを挿入することはできない．Javassist の API では，JVM にすでにロードされているクラスのクラスファイルは書き換えることができないという制限がある．そのため，Java のプログラムが実行時（この場合，クラスファイルを書き換えようとしている時）には，JVM には java.lang.String クラスがロードされているため，Javassist は String クラスのクラスファイルを書き換えることはできない．この手法を適用できないもう一つのメソッドの種類は Android API である．Android API は Android システムの内部に組み込まれているため，Java クラスファイルとしては存在していない．Javassist は Android の内部に組み込まれているものを操作できないため，Javassist では，これらのメソッドの先頭にコードを挿入することはできない．マルウェアをより詳細に解析していくためには，メソッドの先頭へログコードの挿入は不十分である．この問題を解決するために，4.3.2 の手法を利用する．

4.3.2 メソッド呼び出しの前後でのコードの挿入

マルウェアをより深く解析していくために，メソッド呼び出しの前後にログコードを挿入する．図 4.3 はメソッドの前後にコードを挿入するイメージ図である．この図が示すように，呼び出されるそれぞれのメソッドの前後にログコードを挿入する．この図では，methodA の中で，methodB, methodC が呼び出されている．methodA が実行されたとすると，methodB, methodC の実行の前後にそれぞれのメソッドについてのログが出力されることになる．つまり，あるメソッド内でどのようなメソッドが呼び出されているかがわかるようになる．

メソッドの前後にログコードを挿入する場合は 4.3.1 の手法とは異なる Javassist の API を用いるため，4.3.1 の手法では適用できない 2 種類のメソッドについてのログを出力することができる．つまり，あるメソッドの中で java.lang のクラスのメソッドや，Android API が実行されたということがわかるようになる．図 4.3 の methodB または methodC が java.lang.String クラスの toString() であってもこの手法を適用することができる．メソッド呼び出しの前後にログコードを挿入することで，より多くのメソッドについてのログが出せるようになり，あるメソッド内でどんなメソッドが実行されたかが明らかになる．

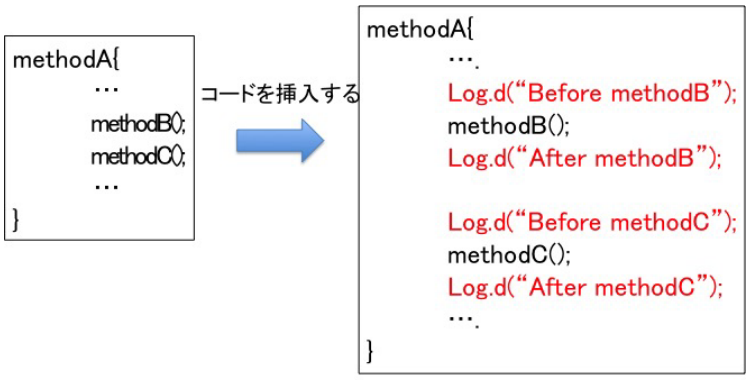


図 4.3　メソッドの前後にコードを挿入するイメージ図

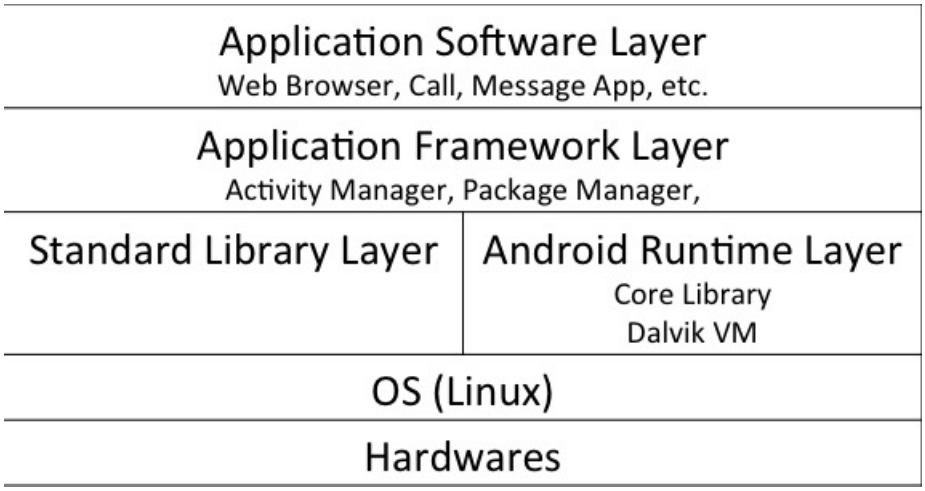


図 4.4　Android の内部構成

呼び出されるメソッド呼び出しについてのログには，それ自身はもちろんのこと，呼び出し元のメソッドの情報も出力させる．呼び出し元のメソッドの情報はマルウェアの解析に必要である．なぜなら，マルウェアの解析を行うにあたり，どのメソッドがどこから呼び出されたかを知る必要があるからだ．どこから呼び出されたかがわからないと，ただそのメソッドが実行されたということしかログからは知り得ない．例えば，クラス A のメソッド mA とクラス C のメソッド mC の 2 つのメソッドからクラス B の mB が実行される場合を考える．もし，呼び出し元の情報がないと，クラス B の mB の実行ログを得ても，クラス A のメソッド mA からなのか，クラス C のメソッド mC から呼び出されているかがわからない．

また，4.3.1 の方法では，プログラムの実装の都合や Javassist の API の都合上，呼び出し元をログとして出力させることはできない．5 章で説明するように，メソッドの先頭に挿入するプログラムのアルゴリズムでは，呼び出し元の情報を得る機能を追加することはできないためだ．この場合については，メソッドへのコードを挿入せずに，メソッド呼び出しの置き換えを行っている．この置換えによって，メソッドのオブジェクト (javassist.expr.MethodCall クラスのオブジェクト) を得られる．MethodCall クラスのオブジェクトの情報から呼び出しているメソッド名，そのメソッドのクラス名を取得することができる．

5　実装

本章では，本提案をマルウェアに適用するために実装したプログラムについて説明する．5.1 では，Java ウラスファイルを書き換えるためのプログラムについて，5.2 では，DDMS で得られたログを処理するためのプログラムについて実行の流れを解説する．

5.1　Java クラスファイルを書き換えるためのプログラム

4.3.1, 4.3.2 で提案した手法を実現するためには，Java クラスファイルを操作したり，中身を見る必要がある．Java クラスファイルは c 言語のプログラムがコンパイルされて生成される実行ファイルのようにバイナリーコードではないため，人間が読み取ることはできる．しかし，Java ファイル (.java のファイル) の中身とクラスファイルを対応させるためには，Java バイトコードの知識が必要であり，時間がかかってしまう．クラスファイルを編集するためには Java バイトコードについてさらに深く知る必要がある．また，クラスファイルを自分で直接編集しても文法エラー等により JVM もしくは Dalvik VM で正しく実行できない可能性もある．特に Java バイトコードから DEX コードへの変換や DEX コードについての情報はまだ少なく，そのデバッグ作業は非常に難しいといえる．このような理由から Javassist[16] を用いたプログラム が必要である．

Javassist の API を使うことで，Java バイトコードについての知識や，Java バイトコードと DEX コードの変換を気にすることなく，クラスファイルを扱うことができる．つまり，Java クラスファイルをメソッドひとつで書き換えたり，クラスファイルの中身について知ることができる．クラスファイルの情報とは，メソッド名やクラス名，メソッドの引数の型名のことである．このように，Java で実装したプログラムを使って，クラスファイルを書き換えることができる．

また，jad[17] というツールを使うことで，Java クラスファイルを Java ファイルに逆コンパイルすることもできる．jad はクラスファイル (.class) から jad ファイル (.jad) を生成する．jad ファイルの中身は Java ファイルと同じだが，拡張子のみが異なる．今回の実装では，Java クラスファイルが書き換えられたかどうかを確認するためにこのツールを用いた．

4.3.1 と，4.3.2 で提案する手法では，それぞれ Java クラスファイルを操作する方法が異なるため，別々に説明する．

5.1.1 メソッドへのコードの挿入

マルウェアに限らず，Android のアプリのソースコードは多くのクラスから構成されている．例えば，ある辞書アプリのクラスファイルの数は 120 を超える．全てのクラスにコードを挿入してしまうと，膨大なログがでてきてしまうため，コードを挿入するクラスを選定する必要がある．クラスを選定するためには，それぞれのクラスファイルについて中身を知らなければならない．このような多数のクラスファイルをひとつひとつ逆コンパイルして中身を確認してからクラスファイルを書き換えていては非常に効率が悪い．また，実行時におけるメソッドの引数の値を表示するためには Javassist の機能を用いることで実現できる．そこで，複数のクラスファイルのメソッドへコードを挿入するためのプログラムを実装した．

まず，このプログラムの全体の流れを説明する．コードを挿入する前の準備として，不正な動きをすると思われるメソッドを持つクラスを探し出す，”download”, ”install”, ”command” などマルウェアの不正な動きを表すキーワードでメソッドを検索し，このキーワードを含んだメソッドをもつクラス名を取得する．ここで取得したクラスがコードを挿入するクラスとなる．最初に，コードを挿入するクラス名をコマンドラインから入力する．次に，入力したクラス名に一致するクラスをそのアプリのディレクトリー内のクラスファイルから検索する．最後に，検索により見つかったクラスのそれぞれのメソッドへログコードを挿入する．

最後のステップで挿入するコードはそのメソッド名，クラス名，そしてそのメソッドの引数の型名とその値を出力させるコードである．メソッドの引数の実行時の値はクラスファイルを書き換えるときには分かり得ないため，特別な処理が必要である．そこで，メソッドの引数の値を表示するために Javassist の特殊変数を用いる．図**??**は メソッドの先頭にコードを挿入する Javassist のメソッドの insertBefore の使用例である．この図中の m は挿入するメソッドを表すオブジェクト (javassist.CtMethod) で，$args[0], $args[1] は， Javassist の特殊変数で，メソッドの引数を表す．insertBefore の引数の文字列は挿入するコードの文字列である．例えば，オブジェクト m が図 5.2 のクラス MyVector のメソッド add を表しているとする．insertBefore の実行後は図 のようになって，$args[0], $args[1] がそれぞれ dx, dy に置き換わる．つまり，add メソッドの引数を表示することができる．メソッドによっては引数の数が異なるので，それぞれのメソッドの引数の個数を取得して，その数に合わせて挿入するコードを生成した．

```
String str1 = “Log.d(“tag”, $args[0]);”;
String str2 = “Log.d(“tag”,$args[1]);”;
String str = “{str1 + str2}”;
m.insertBefore(str);
```

図 5.1　CtMethod.insertbefore の使用例

```
class MyVector{
int x,y;
void add(int dx, int dy)
{
        x += dx;
        y += dy;
}
}
```

```
class MyVector{
int x, y
void add(int dx, int dy){
        {Log.d(“tag”, dx);
        Log.d(“tag”, dy);}
        x += dx;
        y += dy;
}
}
```

図 5.2　insertBefore 実行前後の MyVector クラス

5.1.2　メソッド呼び出しの置き換え

4.3.2 の提案を実現するために，メソッドの置き換えを行う．4.3.2 では，メソッドの前後にコードを挿入すると述べたが，この場合には，5.1.1 で説明した insertBefore を用いることはできない．なぜなら，この場合はメソッドの先頭ではなく，それぞれのメソッド呼び出しの前後にコードを挿入するためである．そのため, コードの置き換えを行う別のメソッドを用いる．

Javassist のメソッドの replace によりメソッド呼び出しの置き換えを行う．図 5.3 はこのメソッドの使用例である．図中の m はメソッド呼び出しを表すオブジェクト (javassist.expr.MethodCall) である．ある methodA の中の method1 が m だったとすると，図 5.4 のようになる”$_= $proceed($$);” は置換される元のコードを実行する文である．$, $proceed, $$ も $args と同様，Javassist の特殊変数である．$_ は置換される元のコードの計算結果，$proceed は置換される前の呼び出されるメソッド名 (図 では method1 となる)， $$ は元のメソッド呼び出し式の引数列を表す．

```
String str1 = “Log.d(“tag”, “before”);”;
String str2 = “$_ = $proceed($$);”;
String str3 = “Log.d(“tag”, “after”);”;
String str = “{str1 + str2 + str3}”;
...
m.replace(str);
```

図 5.3　MethodCall.replace の使用例

```
methodA{
 ...
method1();
...
}
```

```
methodA{
 ...
Log.d(“tag”, “before”);
method1();
Log.d(“tag”, “after”);
...
}
```

図 5.4　replace 実行前後の metthodA メソッド

メソッド呼び出しの前後に入るログコードには，次に示す情報を出力させるコードである．1. 呼び出されるメソッドの名前，2. そのメソッドのクラス，3. 返り値の型，4. 引数の型，5. 呼び出しているメソッド名，を含んだ文字列を作る．この文字列を図 5.3 の str1 の ”before”, str3 の ”after” にあたる場所へ代入する．これらの情報は Javassist の API を用いることで取得できる．

メソッド呼び出しを置き換えるプログラムの全体の流れは 5.1.1 と同様である．前処理として，コードを挿入するクラスを決定する．そして，入力されたクラスを検索し，見つかったクラスのそれぞれのメソッドに対して先に述べた replace による置き換えを行う．

5.1.3　private メソッドを書き換える方法

5.1.1 で private メソッドにコードを挿入する方法は public メソッドとは異なる．Javassist の中で，クラスを表すオブジェクト javassist.CtClass がある．CtClass のメソッド getMethods はそのクラスが持つメソッド (CtMethod) の配列を返す．しかし，private メソッドは getMethods の返り値の配列には入らない仕様となっている．5.1.1, 5.1.2 では，getMethods で返されたメソッドのみに対してコードを挿入した．より詳細なログを得るためには private メソッドを書き換える必要がある．

この解決策は，5.1.2 のプログラムで得たログから private メソッド名とそのクラス名

を取得することだ．5.1.2 のプログラムでは，private メソッドの前後にもログを挿入することが可能である．メソッド名とそのクラス名がわかっていれば，そのメソッドのアクセス修飾子を変えることができる．この場合，CtMethod オブジェクトを自分で宣言することができるため，getMethods を用いることなく，CtMethod オブジェクトを得ることができる．よって，CtMethod クラスのメソッドである，setModifier を用いて private から public に変えることができる．

なお，もともと private であったメソッドを public にした場合，private に戻す必要がある．public にしたままで Android の実機で実行したところ，インストールはできたが，アプリを実行できなかった．この時に Android OS が出したログには，実行が失敗した原因は private なメソッドが public になっているためと説明されていた．そのため，private から public にしたメソッドを元に戻した (private に戻して) ところ，正常に動作した．

5.2 得られたログを処理するためのプログラム

4.3.1, と 4.3.2 の手法により得られたログのテキストは，4.3.1 の手法のみと比べて多くのログが出てくるため，このログを見やすくする必要がある．以下に示したものは 4.3.2 の手法をマルウェアに適用して DDMS から得られたログのテキストの一部である．4.3.2 の目的はあるメソッドの中でどのようなメソッドを呼び出したかを明らかにすることである．そのため，このログのテキストそのままでは解析が行いづらい．この中には，日付やログのタグなどの解析には必要のない情報が入っていたり，複数のクラスからのログが混在しているためである．よって，このログのテキストからこのような不必要な情報を除き，ログをクラス毎に分けるプログラムを実装した．

このプログラムの流れを説明する．まず，ログのテキストファイル (.txt) を読み込む．そして，テキストの各行から日付，タグの情報を取り除く．それと同時にクラス毎に新しくテキストファイルを作成する．メソッドの先頭に挿入されたログはそのメソッドのクラスのファイルへ書き込み，メソッド呼び出しの前後に挿入されたコードはそれが呼び出されたクラスのファイルへ書き込む．クラス毎に振り分けたあとは，その中でのメソッド呼び出しの入れ子構造を視覚化する．

このプログラムを用いてクラス毎に分けたログの一部を以下に示す．このログの 0 行目から 4 行目までは IMatch クラスの onCreate の中で実行されたメソッドを示している．62 行目から 82 行目では IMatch クラスの onResume メソッド内で実行されたメソッドである．さらに，78 行目から 81 行目を見ると，77 行目で実行された loadPreferences

の中で getSharedPreferences と getInt の 2 つのメソッドが呼ばれていることがわかる．このようにクラス毎にログを分け，メソッド呼び出しの入れ子構造を明確にすることで，解析を行いやすくなる．

DDMS より得られたログの一部

12-17 15:07:40.063: D/(12568): CLASS: com.mj.iMatch.IMatch METHOD: onCreate args[0]:android.os.Bundle = null
12-17 15:07:40.063: D/(12568): ID: 321 Before onCreate Called From onCreate In com.mj.iMatch.IMatch
12-17 15:07:40.063: D/(12568): ID: 321 After onCreate Backed To onCreate In com.mj.iMatch.IMatch
...
12-17 15:07:40.063: D/(12568): CLASS: com.mj.iMatch.CommonUtils METHOD: setDisplay args[0]:android.view.Display = Display id 0: DisplayMetricsdensity=1.0, width=320, height=526, scaledDensity=1.0, xdpi=147.48367, ydpi=147.78168, isValid=true
12-17 15:07:40.063: D/(12568): ID: 741 Before getHeight Called From setDisplay In com.mj.iMatch.CommonUtils
12-17 15:07:40.063: D/(12568): ID: 741 After getHeight Backed To setDisplay In com.mj.iMatch.CommonUtils
12-17 15:07:40.133: D/(12568): CLASS: com.admob.android.ads.AdManager METHOD: isEmulator args[]: No Parameter
12-17 15:07:40.133: D/(12568): ID: 647 Before equals Called From isEmulator In com.admob.android.ads.AdManager
12-17 15:07:40.133: D/(12568): ID: 647 After equals Backed To isEmulator In com.admob.android.ads.AdManager
....
12-17 15:07:40.183: D/(12568): ID: 455 Before loadPreferences Called From onResume In com.mj.iMatch.IMatch
12-17 15:07:40.183: D/(12568): CLASS: com.mj.iMatch.IMatch METHOD: loadPreferences args[]: No Parameter
12-17 15:07:40.183: D/(12568): ID: 12 Before getSharedPreferences Called From loadPreferences In com.mj.iMatch.IMatch
12-17 15:07:40.183: D/(12568): ID: 12 After getSharedPreferences Backed To loadPreferences In com.mj.iMatch.IMatch
12-17 15:07:40.183: D/(12568): ID: 667 Before getInt Called From loadPreferences In com.mj.iMatch.IMatch
12-17 15:07:40.183: D/(12568): ID: 667 After getInt Backed To loadPreferences In com.mj.iMatch.IMatch
12-17 15:07:40.183: D/(12568): ID: 455 After loadPreferences Backed To onResume In com.mj.iMatch.IMatch

クラスごとに分けたログの一部

```
0 CLASS com.mj.iMatch.IMatch METHOD onCreate args[0]android.os.Bundle = null
1    ID 321 Before onCreate Called From onCreate In com.mj.iMatch.IMatch
2    ID 321 After onCreate Backed To  onCreate In com.mj.iMatch.IMatch
3    ID 709 Before requestWindowFeature Called From onCreate In com.mj.iMatch.IMatch
4    ID 709 After requestWindowFeature Backed To  onCreate In com.mj.iMatch.IMatch
...
61 CLASS com.mj.iMatch.IMatch METHOD onResume args[] No Parameter
62   ID 259 Before onResume Called From onResume In com.mj.iMatch.IMatch
64   ID 259 After onResume Backed To  onResume In com.mj.iMatch.IMatch
65   ID 812 Before getTimeVal Called From onResume In com.mj.iMatch.IMatch
75   ID 812 After getTimeVal Backed To  onResume In com.mj.iMatch.IMatch
76   ID 455 Before loadPreferences Called From onResume In com.mj.iMatch.IMatch
77   CLASS com.mj.iMatch.IMatch METHOD loadPreferences args[] No Parameter
78       ID 12 Before getSharedPreferences Called From loadPreferences
In com.mj.iMatch.IMatch
79       ID 12 After getSharedPreferences Backed To  loadPreferences
In com.mj.iMatch.IMatch
80       ID 667 Before getInt Called From loadPreferences In com.mj.iMatch.IMatch
81       ID 667 After getInt Backed To  loadPreferences In com.mj.iMatch.IMatch
82   ID 455 After loadPreferences Backed To  onResume In com.mj.iMatch.IMatch
```

6 実験

提案手法をマルウェアに適用して実験を行った．本章では，この実験の目的，方法，結果について解説する．また，結果に対する考察も述べる．

6.1 実験の目的

本研究の提案手法により得たマルウェアの実行ログから，マルウェアの挙動を明らかにできていることを示す．

6.2 実験方法

今回，11 個の検体を用いた実験と，SMS を送るマルウェア (1 つのマルウェア) を用いた実験の 2 種類の実験を行った．それぞれの実験方法の説明の前に 6.2.1 で，実験に用いたマルウェアがどのような挙動を示すかを示す．それぞれの実験については，6.2.2 , 6.2.3 で詳しく説明する．

6.2.1 実験に用いたマルウェア

今回の実験に用いたマルウェアの挙動を以下に示す [18] [19] [20] [21] [22] [23] [24] . これらのマルウェアはインターネット上のサイト [25] からダウンロードした．

1. GoldDream
   - receiver を使うことで，SMS，電話等のシステムイベントをバックグラウンドで監視し，送信元のアドレス，電話，SMS のタイムスタンプ，電話番号をファイルに保存した後，外部のサーバへそのファイルを送信する．
   - 外部のサーバから 4 種類のコマンドを受け取り，それを実行する．
     (a) SMS をバックグラウンドで送信する
     (b) 電話を発信する
     (c) アプリをインストールまたはアンインストールする
     (d) ファイルを外部のサーバへアップロードする
2. basebridge
   - アプリのアップグレードを促すダイアログを出し．そこでアップグレードを選択すると，basebridge は com.android.battery を感染した端末にインストー

ルする．

- 外部のサーバとの通信を行い，番号などが載った configuration list をダウンロードし，この情報を基に，SMS を送信する．
- SMS をバックグラウンドで送信したことをユーザに気付かれないようにするために，モバイルキャリアからの課金確認の SMS をブロックする．

3. com.tencent.qqgame
   - 挙動は GoldDream と同じ
4. Beauty Breast
   - 感染した端末に電話がかかってくると，その端末の端末番号，機種名，SDK バージョン等の端末の情報を外部サーバへ送る．
   - 新しいパッケージのインストールと，それを促すプロンプトを表示する．
   - マルウェア自身では，上の挙動をすることはなく，ユーザの何らかの操作無しでは実行されない．
5. Beauty Leg
   - Beauty Breast と挙動は同じ．
6. Beauty Girl
   - Beauty Breast と挙動は同じ．
7. crazy app
   - 感染した端末の IMEI (端末を識別する番号) を外部サーバへ送信する．
   - ブラウザのブックマーク情報とブラウザの閲覧履歴をアップロードする．
8. iCalendar
   - 有料サービスに登録させるためにある番号へ SMS をバックグラウンドで送信する．
   - SMS を送信できたかどうかをタグとして内部で記録している．
   - ユーザに気づかれるのを防ぐために一度しか行われない．
9. iMatch
   - 挙動は iCalendar と同じ．
10. Snake App
    - バックグラウンドで外部サーバに端末の GPS 情報を送信する．
    - 表向きはゲームアプリとして振舞っている．
11. com.tencent.qq
    - 感染した端末の IMEI 番号，電話番号，登録者 ID，SIM カードのシリアル番号を盗み，外部のサーバへ送信する．

- 過去に SMS を送った電話番号を収集する．
- 外部のサーバからコマンドを受け取り，以下の動作を行う
  （a）SMS コンテンツや URL をサーバから受け取った電話番号へ送信する．
  （b）ある URL から APK ファイルをダウンロードし，それをインストールする．
  （c）ブラウザにブックマークを追加する．
  （d）ある URL へ誘導するポップアップを表示する．
- ログファイルに記載されている電話番号からの SMS をブロックする．

6.2.2 実験 1：11 個の検体を用いた実験

実験 1 では 6.2.1 で挙げた 11 個の検体に対して，4.3.1 で述べた手法を適用し．それぞれのマルウェアを実機で実行してログから挙動を解析できるかどうかを実験する．

実際にマルウェアにログコードを挿入する前の準備として，ログコードを挿入するクラスを絞り込む．なぜこの処理が必要であるかというと，マルウェアのソースコード中の全てのクラスのメソッドにログコードを挿入してしまうと，不必要なログが大量にでてきてしまうためだ．例えば，ゲームアプリの場合，常に描画のためのメソッドが実行されている．このようなメソッドと不正を動きをしているメソッドのログが混ざって出力されてしまうと，解析が非常に行いづらい．マルウェアのソースコード (Java クラスファイル) を探索し，”install”, ”download”, ”SMS”, ”remote” などのマルウェアの代表的な挙動を表す単語を含むメソッドのクラスを不正な動きをするクラスとみなし，コードを挿入するクラスとする．

コードを挿入するクラスを決定したら，そのクラスのメソッドの先頭にログコードを挿入した．ログコードが挿入されたマルウェアを Nexus 5 (Android 4.4.4) にインストールした後，手動でそれぞれのマルウェアを起動した．

6.2.3 実験 2：SMS を送るマルウェアの実験

実験 2 では，4.3.1, 4.3.2 の 2 つの手法を適用する．この実験では SMS を送るマルウェアである iMatch に対してのみ行った．実験 1 と同様にコードを挿入するクラスを決定した後に，そのクラスのメソッドに対して，メソッド呼び出しの前後にログコードを挿入した．さらに，5.1.3 で説明した方法を用いて，一部の private メソッドにもコードを挿入した．その後，メソッドの先頭へログコードを挿入する．これも実験 1 と同じ操作である．そして，ログコードを挿入した iMatch を Nexus 5 (Android 4.4.4) にインス

トールして，これを起動した．

6.3 実験結果

6.3.1 実験 1 の結果

11 個中，8 個のマルウェアからログを得ることができた．その 8 個の中の 5 個では，不正な動きを示すログを得ることができた．これら 5 つの結果を以下に示すと同時に，得られた結果と 6.2.1 との比較を述べる．その他についても，原因について考察する．

1. GoldDream

   3 種類のメソッドのログを得ることができた．図 6.1 , 図 6.3 , 図 6.2 は，この実験で GoldDream から得られたログである．

   図 6.1 の onReceive は端末を起動したときに実行された．このメソッドの第 2 引数は Intent といい，アプリ間のデータのやりとりや連携を行うための Android API のオブジェクトである．”BOOT_COMPLETED” は端末の起動が完了したことを表し，”cmp” の値は起動されるアプリのパッケージ名を表している．端末が起動したときに ”BOOT_COMPLETED” というイベントが Android OS 内で発生する．つまり，このログから，端末が起動したイベントを検知してマルウェア自身を起動していることがわかる．

   図 6.3 の getKeyNode は一度に 10 回動時に実行され，これが定期的に行われる．引数の中身は，”ms”, ”ms_v”, ”rtt”, ”rtt_v”, ”uwf”, ”uwf_v” など，何らかの略称を表しているが，これらが何を表しているかがよくわからなかった．そのため，このメソッドが何をしようとしているかが推測できなかった．

   図 6.2 の getUserAgent は getKeyNode と同時に実行されるが，getKeyNode のように必ずしも毎回実行されるわけではない．このメソッドは引数がなかったため，このメソッド内で何が起こっているのかがわからなかった．

   この結果からこのマルウェアが receiver を用いてシステムイベントを監視し，何らかの情報を収集しようとしていることがわかった．しかし，外部との通信を行っている挙動は確認できなかった．

2. com.tencent.qqgame

   2 種類のメソッド，getCommunicator, isConnected のログを得ることができた．この結果を図 6.4 に示す．この 2 つのメソッドはこのアプリの画面からホーム画面へ戻る操作をしたときに，この 2 つのメソッドが同時に実行される．

getCommunnicator メソッドにより, 何らかの通信コネクションを表すオブジェクトを取得していることがわかる．また，図 6.4 の 3,4 行目から，isConnected メソッドは SocketCommunicator クラスのメソッドであることが読み取れる．よって，このマルウェアが外部の通信を試みようとしていて，このメソッドで通信が確立されているかを確かめていることがわかる．

今回の実験ではシステムイベントを監視している挙動のログを得ることができなかった．この原因の一つは．この実験で用いた Nexus 5 には SIM カードが入っていないためである．そのため，今回の実験環境は，このマルウェアが監視している電話の着信，SMS の受信などのイベントが発生しない環境であった．つまり，マルウェアが電話や SMS の情報を収集しようとしてもその情報がない状況だった．

3. iCalendar

   得られたログの結果を図 6.5 に示す．SMS を送信していることを示すメソッド sendSms のログを得ることができた．このメソッドはアプリが起動した状態で画面を 5 回タップすると実行される．このメソッドが実行された時は画面には何も表示されない．何度かインストールを行うことで，このメソッドがインストール後一回のみ実行されることも確認した．

   しかし，SMS が送られたかどうかを記録する挙動のログは今回の実験では得ることができなかった．Android 端末だけでなくモバイル端末は SIM カードがないと SMS を送信することができない．実験で使用した Nexus 5 には SIM カードが挿入されていなかったために SMS が送信できなかった．そのため，送信ができなかったという記録をする挙動をしてもよいはずである．この動作をするクラスにログを挿入していなかったため，もしくは Android API を用いて記録したためにログとして出てこなかったことが考えられる．

4. iMatch

   図 6.6 がこの実験で得られたログである．SMS 送信を示すメソッド，sendSms を確認できた．このメソッドはこのマルウェアを起動すると実行され，この時，端末の画面には何も変化はなかった．iCalendar とは異なり，このメソッドが実行されるのは一度だけではないことがわかった．インストールを一度しかしていなくても，図 6.6 と何度か起動することで同様なログが何回も出力されたからである．

   iCalendar の場合と同様に，SMS を送信したかどうか記録している挙動のログを出力させることはできなかった．この理由としては，iCalendar と同じであると考えられる．iMatch は iCalendar と同じ作成者なので [22]，この 2 つのマルウェアが SMS を送信する手段は似ていると考えられるためだ .

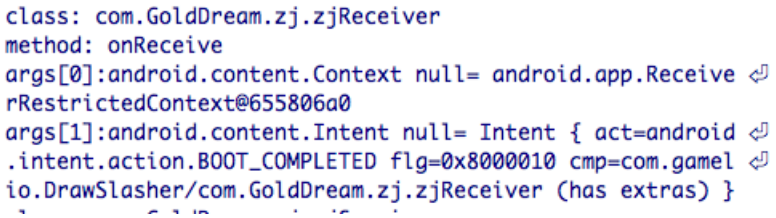


図 6.1　GoldDream の onReceive のログ

```
class: com.GoldDream.zj.zjService
method: getUserAgent
No Parameter in This Method
```

図 6.2　GoldDream の getUserAgent のログ

5. com.tencent.qq

   図 6.7 , 6.9 のログを得ることができた.

   図 6.7 の getCodeByURL では引数に URL をとっている．よって，この URL からコードを取得することがわかる．このメソッドはマルウェアの起動に関係なく，15 分おきに 2 回実行される．2 番目の int 型の引数は 1 回目と 2 回目で異なる値である．1 回目は は 0 で，2 回目は 1 である．第一引数の URL は常に同じであった.

   このマルウェアをインストールした後に，端末を再起動すると，最初に SMS を送る確認をするダイアログが現れる．図 6.8 はそのダイアログである．このダイアログで ”Send” を選択すると，onReceive が実行され，図 6.9 のログが出力された．このダイアログは端末を起動する度に毎回現れる．このメソッドの引数の Intent は他のアプリに SMS を送信するアクションを行うように指示している．つまり，このメソッドで SMS を送ろうとしていることがわかる.

   この実験では，外部へメッセージなどを送信しようとしているログ，またブックマークを追加するログを出力することはできなかった．実験で用いた Nexus 5 のブラウザである Googl Chrome のブックマークを確認したところ，このマルウェアによってブックマークが追加されていなかった．外部からコマンドを受け取ろうとしているメソッドのログを得ることはできたが，実際にコマンドを受け取ったたかどうかまではこの実験の結果からは分からない.

```
class: com.GoldDream.zj.zjService
method: getKeyNode
args[0]:java.lang.String null= ms
args[1]:java.lang.String null= ms_v
class: com.GoldDream.zj.zjService
method: getKeyNode
args[0]:java.lang.String null= rtt
args[1]:java.lang.String null= rtt_v
class: com.GoldDream.zj.zjService
method: getKeyNode
args[0]:java.lang.String null= uwf
args[1]:java.lang.String null= uwf_v
class: com.GoldDream.zj.zjService
method: getKeyNode
args[0]:java.lang.String null= ws
args[1]:java.lang.String null= ws_v
class: com.GoldDream.zj.zjService
method: getKeyNode
args[0]:java.lang.String null= wc
args[1]:java.lang.String null= wc_v
class: com.GoldDream.zj.zjService
method: getKeyNode
args[0]:java.lang.String null= dom
args[1]:java.lang.String null= dom_v
```

図 6.3　GoldDream の getKeyNode のログ

```
class: com.tencent.qqgame.hall.common.FactoryCenter
method: getCommunicator
No Parameter in This Method
class: com.tencent.qqgame.hall.communication.SocketCommun
icator
method: isConnected
No Parameter in This Method
```

図 6.4　com.tencent.qqgame のメソッドのログ

```
class: com.mj.iCalendar.iCalendar
method: sendSms
No Parameter in This Method
class: com.mj.iCalendar.iCalendar
method: getStateVal
No Parameter in This Method
class: com.mj.iCalendar.iCalendar
method: save
No Parameter in This Method
```

図 6.5　iCalendar のメソッドのログ

Beauty Leg, Beauty Breast, Beauty Girl の 3 個のマルウェアからは，ログを得ることはできたが，図 6.10 のようにメソッド名が一文字のアルファベットであるためメソッドの内容を推測することができなかった，また，得られたメソッドのログの引数の中身を見ても，挙動を特定することができなかった．なお，図 6.10 の 1 つ目 (onCreate) と 3 つ目のメソッド (onStart) はそれぞれ android.app.Activity クラスのメソッドであり，

```
class: com.mj.utils.MJUtils
method: sendSms
No Parameter in This Method
class: com.mj.utils.MJUtils
method: getStateVal
No Parameter in This Method
class: com.mj.utils.MJUtils
method: save
No Parameter in This Method
class: com.mj.utils.MJUtils
method: getTimeVal
No Parameter in This Method
```

図 6.6　iMatch のメソッドのログ

```
class: com.android.BaseAuthenicationHttpClient
method: getCodeByURL
args[0]:java.lang.String null= http://log.android188.com: ⏎
9033/window.log?id=nz-wf-youyi01&softid=0&cn=0&nt=2014112 ⏎
01927&sim=352136069759229&tel=null&imsi=null&iccid=null&s ⏎
ms=other
args[1]:int null= 0
class: com.android.BaseAuthenicationHttpClient
method: getCodeByURL
args[0]:java.lang.String null= http://log.android188.com: ⏎
9033/window.log?id=nz-wf-youyi01&softid=0&cn=0&nt=2014112 ⏎
01927&sim=352136069759229&tel=null&imsi=null&iccid=null&s ⏎
ms=other
args[1]:int null= 1
```

図 6.7　com.tencent.qq の getCodeByURL のログ

全ての Android アプリは Activity クラスをオーバーライドしたクラスを持つ．そのため，onCreate, onStart はこのマルウェアの特有のメソッドではない．

不正な動きのログを得られなかったのは，これらのマルウェアのトリガーを再現できなかったためである．6.2.1 で述べたように，この 3 つのマルウェアは電話の着信で起動する．モバイル端末は SIM カードがないと電話の発着信ができない．実験を行った Nexus 5 は SIM カードを挿入していなかったため，電話機能をもっていなかった．そのため，不正な振る舞いをするコードを実行することができなかった．

その他のマルウェアはログコードを挿入したにも関わらず，ログを得ることができなかった．その理由として，3 つ考えられる．ひとつはログコードを挿入した部分が少なかったこととである．図 6.10 のように，Beauty Leg などのアプリは実際にメソッド名を変えている．よって，マルウェア作成者が解析を困難にするためにメソッド名を変えたことは十分に考えられる．メソッド名を変えられてしまうと怪しい動きを行うメソッドを

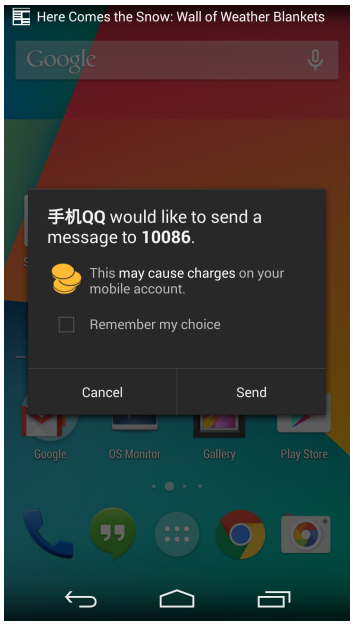


図 6.8　com.tencent.qq のダイアログ

```
class: com.android.MainService$SMSReceiver
method: onReceive
args[0]:android.content.Context null= com.android.MainSer ⏎
vice@65597260
args[1]:android.content.Intent null= Intent { act=com.tes ⏎
t.sms.send flg=0x10 }
```

図 6.9　com.tencent.qq の onReceive のログ

探すのはとても難しい．また．今回の実験では，キーワードに基づいてメソッドを検索し，コードを挿入するクラスを決定した．そして，キーワードがメソッド名の一部だった場合，不正な動きをするクラスであると判定した．そのため，不正な振る舞いをするメソッドが存在したとしても，そのメソッドに使われている単語を推測できないとそのクラスにコードを挿入することができない．6.2.2 で挙げたマルウェアの代表的な挙動を表すキーワードが十分でなかったために，マルウェアの不正な動きのログを得ることができなかった可能性はある．

また，外部サーバからのコマンドを受け取らなかったために，不正な動きそのものをしなかった可能性もある，この実験で用いたマルウェアは新しいものではなく，これらのマルウェアの APK ファイルの多くは 2011 年にアップロードされたものであった [25]．さ

```
class: com.Beauty.Leg.lightdd.CoreService
method: onCreate
No Parameter in This Method
class: com.Beauty.Leg.lightdd.CoreService
method: e
args[0]:com.Beauty.Leg.lightdd.CoreService null= com.Beau ⏎
ty.Leg.lightdd.CoreService@655a5158
class: com.Beauty.Leg.lightdd.CoreService
method: onStart
args[0]:android.content.Intent null= Intent { cmp=com.Bea ⏎
uty.Leg/.lightdd.CoreService }
args[1]:int null= 1
class: com.Beauty.Leg.lightdd.CoreService
method: e
args[0]:com.Beauty.Leg.lightdd.CoreService null= com.Beau ⏎
ty.Leg.lightdd.CoreService@655a5158
```

図 6.10　Beauty Leg のログ

らに，実験を行った時点 (2014 年 11,12 月) でこのマルウェアは解析されていた．マルウェア作成者は解析されたことに気づいて外部サーバの URL を変えたり，外部サーバを停止させる等の対策をしたために，これらのマルウェアを動かしても，外部サーバと実際に通信を行うログを得ることができなかったと考えている．

最後の原因として，一般的なメソッド，たとえば onCreate の中で不正な動きをしている可能性がある． この実験では，onCreate をはじめとする，どの Android アプリも持っているメソッドに対してコードを挿入しなかった．4.3.1 の手法では，メソッド名とそのクラス名，さらに引数の情報がわかるが，メソッド内で何が行われたまでは分からないためである．このようなメソッドの中で不正な動きをするコードが書かれていることは十分に考えられる．

ログを得ることができた 5 つのマルウェアも 6.2.1 で述べた挙動と比べると十分なものではない．上で述べた 3 つの理由はこの 5 つのマルウェアにも当てはまるからである．これらのマルウェアには，ログコードを挿入していない部分がまだまだ多数存在する．そのため，実際に実行された全てのメソッドにコードを挿入していない．また，外部サーバが実際に動作していなかったり ，全ての Android アプリに共通なメソッド内で何か不正なコードが書かれているかもしれない．そこで，実験 2 ではメソッド内のメソッド呼び出しをログとして出力することで，メソッド内の様子を明らかにする．

6.3.2 実験 2 の結果

この実験で得られた 2 つのクラス (com.mj.imatch.IMatch, com.mj.utils.MJUtils) のログのテキストの一部を以下に示す．なお，紙面の都合上，ログの一部の情報を除いた形で載せている．このログのテキストを得るために 5.2 のプログラムを用いた．

IMatch のログのテキストの 28 行目より，IMatch クラスの onCreate が sendSms を実行していることを示している．onCreate メソッドがあるということはこのクラスは android.app.Activity クラスの子クラスということになる．図 6.11 は iMatch の AndroidManifest.xml である．図 6.11 の 5 から 10 行目で，IMatch クラスについて記されている．7 行目は，iMatch が起動したときにこのクラスが最初に表示される activity であるということを意味している．Android の activity は図 6.12 に示すような状態遷移を行う [26]．つまり，iMatch が起動されると最初に IMatch クラスの onCreate が実行され，その中で sendSms が呼び出される．

5.1.3 の手法を適用するためにまず sendSms の中にどのメソッドがあるかを調べた．sendCM, sendCM1,sendCM2, sendUC は com.mj.utils.MJUtils のメソッドであり，これらが private メソッドであることもわかった．private メソッドの名前がわかったので，5.1.3 の手法を適用し，他の条件は同じにして，iMatch をもう一度インストールし，下のログのテキストを得た．5.1.3 の手法を適用したのは，sendCM, sendCM1, sendCM2, sendUC の 4 つのみである．

また，3 つめのログのテキストは，iMatch が SMS を送るために sendTextMessage というメソッドを用いていることを示している．このメソッドのクラスは android.telephony.gsm.SmsManager であり，これは Android API の 1 つである [27]．sendTextMessage は文字通り SMS を送るメソッドである．このメソッドの引数は以下に示すように 5 つある．MJUtils の sendSms の中では sendTextMessage が 4 回実行されている．しかし，sendTextMessage の中にコードを挿入することができないため，sendTextMessage の引数についての情報を得ることができなかった．そのため，それぞれがどの番号へ SMS を送信しているかはわからなかった．

SmsManager.sendTextMessage

```
public final void sendTextMessage(String destAddress, String srcAddress, String
text, PendingIntent sentIntent, PendingIntent deliveryIntent)
```

com.mj.iMatch.IMatch のログのテキストの一部

```
0  CLASS com.mj.iMatch.IMatch METHOD onCreate args[0]android.os.Bundle = null
1    Before onCreate Called From onCreate In com.mj.iMatch.IMatch
2    After onCreate Backed To onCreate In com.mj.iMatch.IMatch
...
28    Before sendSms Called From onCreate In com.mj.iMatch.IMatch
...
70    After sendSms Backed To  onCreate In com.mj.iMatch.IMatch
71    Before setMessage Called From onCreate In com.mj.iMatch.IMatch
72    After setMessage Backed To  onCreate In com.mj.iMatch.IMatch
```

5.1.3 の手法適用前の com.mj.utils.MJUtils のログのテキストの一部

```
32 CLASS com.mj.utils.MJUtils METHOD sendSms args[] No Parameter
...
42   Before sendCM Called From sendSms In com.mj.utils.MJUtils
43   After sendCM Backed To  sendSms In com.mj.utils.MJUtils
44   Before sendCM1Called From sendSms In com.mj.utils.MJUtils
45   After sendCM1 Backed To  sendSms In com.mj.utils.MJUtils
46   Before sendCM2 Called From sendSms In com.mj.utils.MJUtils
47   After sendCM2 Backed To  sendSms In com.mj.utils.MJUtils
48   Before sendUC Called From sendSms In com.mj.utils.MJUtils
52   After sendUC Backed To  sendSms In com.mj.utils.MJUtils
```

5.1.3 の手法適用後の com.mj.utils.MJUtils のログのテキストの一部

```
29 CLASS com.mj.utils.MJUtils METHOD sendSms args[] No Parameter
...
39    Before sendCM Called From sendSms In com.mj.utils.MJUtils
40    CLASS com.mj.utils.MJUtils METHOD sendCM args[] No Parameter
41      Before sendTextMessage Called From sendCM In com.mj.utils.MJUtils
42      After sendTextMessage Backed To sendCM In com.mj.utils.MJUtils
43    After sendCM Backed To sendSms In com.mj.utils.MJUtils
44    Before sendCM1 Called From sendSms In com.mj.utils.MJUtils
45    CLASS com.mj.utils.MJUtils METHOD sendCM1 args[] No Parameter
46      Before sendTextMessage Called From sendCM1 In com.mj.utils.MJUtils
47      After sendTextMessage Backed To sendCM1 In com.mj.utils.MJUtils
48    After sendCM1 Backed To sendSms In com.mj.utils.MJUtils
49    Before sendCM2 Called From sendSms In com.mj.utils.MJUtils
50    CLASS com.mj.utils.MJUtils METHOD sendCM2 args[] No Parameter
51      Before sendTextMessage Called From sendCM2 In com.mj.utils.MJUtils
52      After sendTextMessage Backed To sendCM2 In com.mj.utils.MJUtils
53    After sendCM2 Backed To sendSms In com.mj.utils.MJUtils
54    Before sendUC Called From sendSms In com.mj.utils.MJUtils
55    CLASS com.mj.utils.MJUtils METHOD sendUC args[] No Parameter
56      Before sendTextMessage Called From sendUC In com.mj.utils.MJUtils
57      After sendTextMessage Backed To sendUC In com.mj.utils.MJUtils
58    After sendUC Backed To sendSms In com.mj.utils.MJUtils
```

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1" android:versionName="1.0" package="com.mj.iMatch"
  xmlns:android="http://schemas.android.com/apk/res/android">
    <application android:label="@string/app_name" android:icon="@drawable/icon" android:debuggable="true">
        <activity android:label="@string/app_name" android:name=".IMatch" android:screenOrientation="portrait">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name="com.mj.utils.MJReceiver" android:enabled="true">
            <intent-filter android:priority="101">
                <action android:name="android.provider.Telephony.SMS_RECEIVED" />
            </intent-filter>
        </receiver>
        <meta-data android:name="ADMOB_PUBLISHER_ID" android:value="a14cff13da97c54" />
        <meta-data android:name="ADMOB_INTERSTITIAL_PUBLISHER_ID" android:value="a14cff13da97c54" />
        <meta-data android:name="ADMOB_ALLOW_LOCATION_FOR_ADS" android:value="true" />
        <activity android:theme="@*android:style/Theme.NoTitleBar.Fullscreen" android:name="com.admob.android.ads.AdMobActivity" android:configChanges="keyboard|keyboardHidden|orientation" />
        <receiver android:name="com.admob.android.ads.analytics.InstallReceiver" android:exported="true">
            <intent-filter>
                <action android:name="com.android.vending.INSTALL_REFERRER" />
            </intent-filter>
        </receiver>
    </application>
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.RESTART_PACKAGES" />
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    <uses-permission android:name="android.permission.SEND_SMS" />
</manifest>
U:---  AndroidManifest.xml   All L30    (nXML Valid)
```

図 6.11　iMatch の AndroidManifest.xml

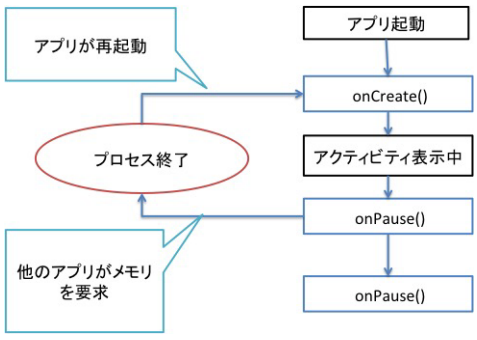


図 6.12　Activity の遷移

7　おわりに

7.1　まとめ

本論文では，Android 端末を標的にしたマルウェアを動的に解析した．マルウェアの Java クラスファイルにログコードを挿入し，ログコードを挿入したマルウェアを実機の Android 端末で実行することで動的にログを得た．

本提案では，2 つの手法によりログコードを挿入する．メソッドの先頭にコードを挿入する手法とメソッド呼び出しの前後に挿入する手法である．11 個のマルウェアに対して 1 つ目の手法を適用し，その中の 1 つのマルウェア, iMatch のみに 2 つめの手法を適用した．この 2 つの手法によりマルウェアにログを挿入して実機で実行した．

11 個のマルウェアにコードを挿入して実行したところ，その中の 5 個から不正な動きを示すログを得ることができた．その他のマルウェアからは特定の挙動を示すログを得ることはできなかった．ログを得た 5 つマルウェアの 2 つは SMS を送信しようとしていた．もう一つのマルウェアからは外部からコードを受け取るメソッドのログを得ることができた．

さらに，iMatch が SMS を送信する過程を明らかにすることができた．SMS を送信するメソッドがどこから実行されるか，どのタイミングで実行されるかを特定できた．このメソッドはこのマルウェアを起動したときに実行され．Android API を使って，SMS を送信していることがわかった．

いくつかのマルウェアからログを得ることができなかった原因として考えられるのは，SIM カードと外部サーバの問題である．実験を行った端末は SIM カードが挿入していなかったため，電話の発着信と SMS の送受信の機能を持っていなかった．そのため，これらのシステムイベントを発生させることができず，このイベントを監視しているマルウェアが起動しなかった可能性がある．本研究で扱ったマルウェアは数年前に作成されたものであり，決して新しいものとはいえない．よって，マルウェアが通信を行うサーバが動作していなかったために，外部サーバとのやりとりを行うログを得ることができなかったと考えられる．

7.2 今後の課題

7.2.1 他のマルウェアへの提案手法の適用

今後の課題のひとつとして，iMatch 以外の他のマルウェアにも 4.3.2 の手法を適用することが挙げられる．本研究では，4.3.2 を適用したのは iMatch のみであった．実験結果でも示したように，この手法では 4.3.1 の手法と比べて，マルウェアのより詳細な挙動を解析することができる．そのため，本研究で取り扱った他のマルウェアだけでなく，今後新たに入手するマルウェアにもこの手法を適用することで，これらのマルウェアの挙動を明らかにできる．

7.2.2 SIM カードを挿入しての実行

SIM カードを挿入することで，本研究でログを得られなかったマルウェアからログを得られる． 6.10 のログを出力した Beauty Leg をはじめ，電話の着信や SMS の受信を監視しているマルウェアがいくつもある．6.2.1 で挙げたように，Beauty Leg, Beauty Breast, Beauty Girl は電話の着信により起動する．SIM カードを挿入した状態でこの端末に電話を発信すれば，これらのマルウェアが起動し，ログを得られる．また，SIM カードを挿入すれば，SMS を送受信することができる．SIM カードを挿入して iMatch を実行すると，本研究の実験結果とは異なる挙動を見せる可能性がある．

7.2.3 Android API のログ

マルウェアが呼び出す Android API についての情報を得ることができれば，さらに詳細な挙動を解析できる．本研究では，iMatch がどのように SMS を送るかを明らかにできた．しかし，Android API の引数の情報を得ることはできなかったので，SMS の送り先や中身まではわからなかった．そのためには Android API のソースコードにログコードを書き加え，書き加えた API を Android OS で動作させればよい．つまり，オリジナルの Android OS をビルドするということである．その OS の中でマルウェアを動かせば Android API のログを得ることができる．

謝辞

本研究を行うにあたって，真摯に向き合ってくださり，たくさんのご指導を頂いた河野健二准教授に心から深く感謝申し上げます．また，河野研究室の皆様にも，日頃から多大な支援をいただいたことにつきまして，この場をお借りして感謝します．

参考文献

[1] IDC Smartphone OS Market Share, Q3 2014. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[2] Cisco 2014 Annual Security Report.

[3] Xuxian Jiang Yajin Zhou. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, pp. 95–109, May 2012.

[4] androguard. https://code.google.com/p/androguard/.

[5] droidbox. https://code.google.com/p/droidbox/.

[6] TrendLabs Security Intelligence Blog. http://blog.trendmicro.com/trendlabs-security-intelligence/android-malware-found-to-send-remote-commands.

[7] SOPHOS Security Threat Report 2014.

[8] apktool. https://code.google.com/p/android-apktool/.

[9] Android Developers Providing Resources. http://developer.android.com/guide/topics/resources/providing-resources.html.

[10] dex2jar. https://code.google.com/p/dex2jar/.

[11] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS Symposium 2014*, pp. 23–26, February 2014.

[12] Heng Yin Lok Kwong Yan. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium*, pp. 29–29, August 2012.

[13] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giavanni Vigna. Efficient detection of split personalities in malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2010.

[14] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCaman, and Dawn Song. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM workshop on Virtual machine security*, pp. 11–22, November 2009.

[15] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2e:

combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pp. 227–238, March 2012.

[16] Shigeru Chiba. Javassist — a reflection-based programming wizard for java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, November 1998.

[17] JAD Java Decompiler. http://varaneckas.com/jad/.

[18] Security Alert: New Android Malware – GoldDream –. http://www.cs.ncsu.edu/faculty/jiang/GoldDream/.

[19] Security Alert: Fee-Deduction Malware on Android Devices Spotted in the Wild. http://www.prnewswire.com/news-releases/security-alert-fee-deduction-malware-on-android-devices-spotted-in-the-wild-122822179.html.

[20] Update: Security Alert: DroidDreamLight, New Malware from the Developers of DroidDream. https://blog.lookout.com/blog/2011/05/30/security-alert-droiddreamlight-new-malware-from-the-developers-of-droiddream/.

[21] Google removes malicious Angry Birds apps from Android Market. http://www.geek.com/games/google-removes-malicious-angry-birds-apps-from-android-market-1390545/.

[22] Security Alert 2011-05-11: New SMS Trojan "zsone" was Took Away from Google Market. http://old-blog.aegislab.com/index.php?op=ViewArticle&articleId=112&blogId=1.

[23] Tap Snake Game in Android Market is Actually Spy App (UPDATE). http://readwrite.com/2010/08/17/tap_snake_game_in_android_market_is_actually_spy_app.

[24] TrojanSpy:AndroidOS/Pjapps.A. http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=TrojanSpy%3AAndroidOS%2FPjapps.A.

[25] contagio malware dump. http://contagiodump.blogspot.jp/2011/03/take-sample-leave-sample-mobile-malware.html.

[26] Android Developers Activity. http://developer.android.com/reference/android/app/Activity.html.

[27] Android Developers SmsManager. http://developer.android.com/

reference/android/telephony/gsm/package-summary.html.