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ることがその理由の１つである．様々なメーカーによって開発が行われ，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の増加から，Android マルウェアが急激に増加したことを報告している．具体的には，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

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のみを赤く表示する．もし，マルウェアが外部から攻撃コードをダウンロードするという攻撃をする場合，静的解析では，対応することはできない．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からだ．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．それを用いてマルウェアの解析を行う．

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，例えば，SMS の送信や，外部からのコードの入手を示していた．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，ある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，これの攻撃手段である，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基本的なについて解説する．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 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，特権レベルを上げることだ．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 の挙動を示す．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．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 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．図 2.1 に AndroidManifst.xml の例を示す．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．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 com.facebook.katana は は com.instagram.android，となっている．一般的に使用している分には，ユーザは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．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．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．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．

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

```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 2.1　AndroidManifest.xml の例

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 [9] 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．

3 関連研究

Android マルウェアを解析，調査した研究を 3 つ紹介する．

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．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かっており，外部サーバのアドレスの格納方法がジョジョジョに複雑になっている．AnserverBot は解析回避と遠隔操作の 2 つの特徴を持つ．さらに，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，分類は行わず，解析のみを行っている．

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解析を行っている [10] ． 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．この研究での外部コードのローディングを検知するツールは，静的に行っているため，マルウェアを全く動かしていない．それに対して本研究ではマルウェアを実機で実際に動かして動的にログを得ることで解析している．また，この研究では，ひとつの攻撃手法に限定して解析を行っているが，本研究では，特定の攻撃手法に限定していない．

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 [11] ．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．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．そのため，

実機で解析すればマルウェアの ”正常” な動作を解析することができる．

ここで紹介した研究には問題点が残っている．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， reflection を通して実行することが考えられる．S.Poelau 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．DroidScope は動的解析のため，コードカバレッジが制限される．実行時には，1 つの実行パスしか通ることはない．L.Yan, H.Yin 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．

4 提案

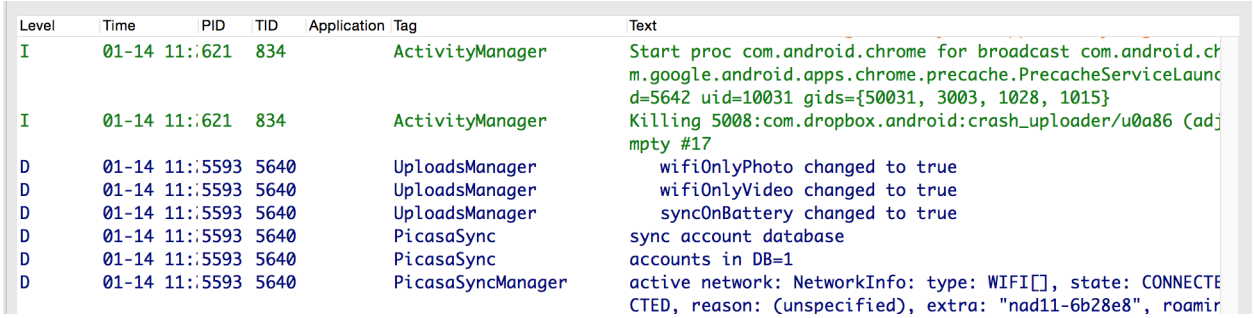
本研究では，Android を標的にしたマルウェアの動的解析を行う．マルウェアにログコードを挿入させ，そのログを動的に得ることで解析を行う．エミュレータでは再現できないために解析できないことも，本提案を適用することで，実機でマルウェアを動かすため解析できるメリットがある．4.1 では，本提案の全体の流れを説明する．4.2 では，本提案の解析手順を，4.3 では，ログコードをどこに挿入するかについて説明する．

4.1 全体の流れ

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 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 からわかるように，それぞれのログにはタグがある．そ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

まず最初に，APK ファイルから Java クラスファイルを取り出す．**??** 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 [9] 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



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

で Android アプリの Java クラスファイルを得る．

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 [12] 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．

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”
```

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．

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．

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

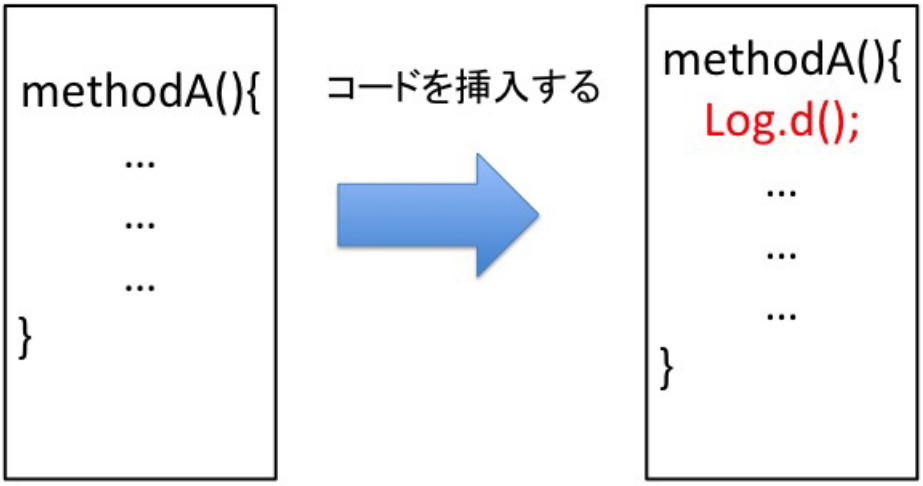


図 4.2　メソッドの先頭へコードを挿入するイメージ図

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 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 4.2 を以下に示す．図 4.2 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．

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．methodA の中で，methodB, methodC が呼び出されるとする．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．マルウェアをより詳細に解析していくためには，4.3.1 で述べた，メソッドの先頭へログコードの挿入は不十分である．なぜなら，全てのメソッドに適用することはできないためだ．適用できないメソッドは 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．4.3.1 の方法が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．しかし，メソッドの前後にログコードを挿入する場合は Javassist の異なる API を用いるため，この 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， String クラスのメソッドや，Android API が実行されたというこ

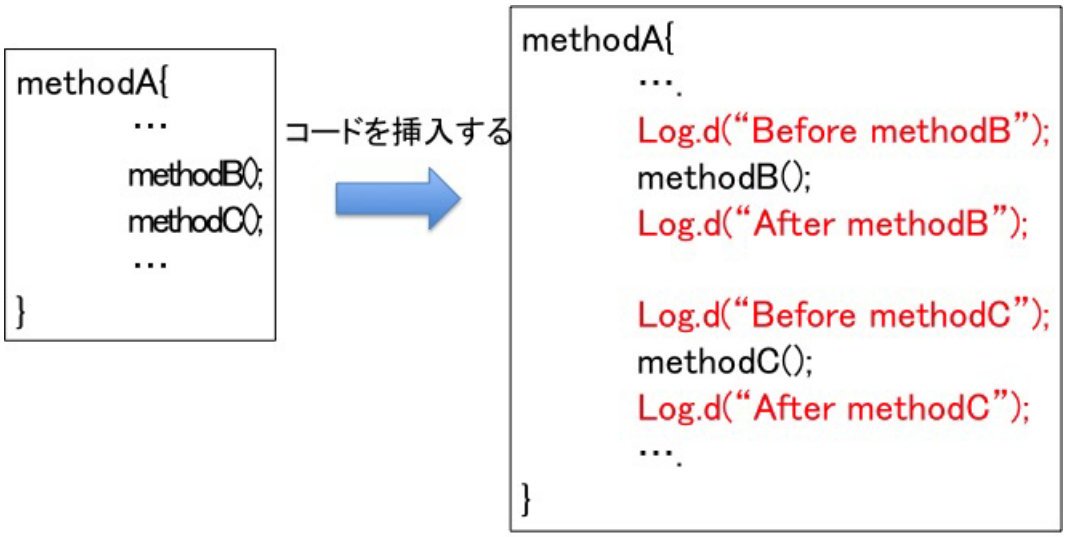


図 4.3　メソッドの前後にコードを挿入するイメージ図

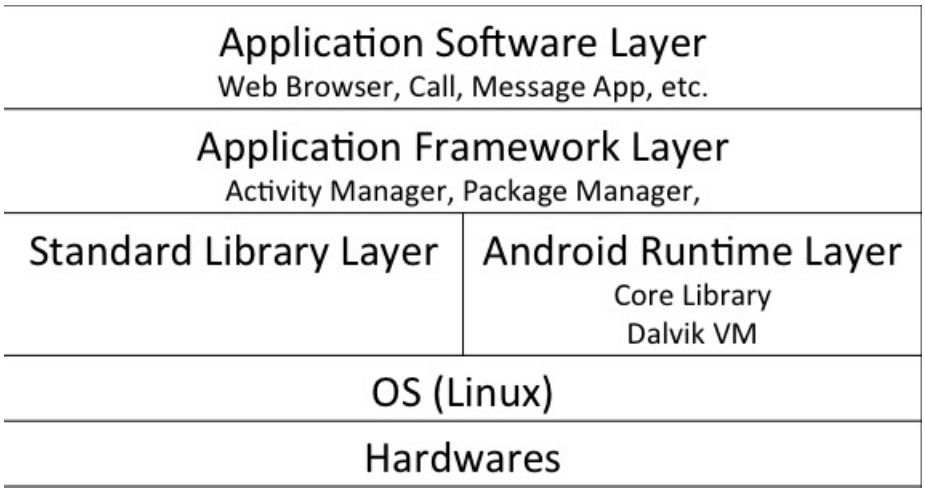


図 4.4　Android の内部構成

とがわかるようになる．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．

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．メソッドの前後にコードを挿入する，と先に書いたが，実装上はメソッド呼び出しの置き換えを行っている．この置換えの際に呼び出される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

5.1 Java クラスファイルを書き換えるためのプログラム

5.1.1 メソッドへのコードの挿入

5.1.2 メソッド呼び出しの置き換え

5.1.3 private メソッドを書き換える方法

5.2 得られたログを処理するためのプログラム

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．実験の説明の前に 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 [13] [14] [15] [16] [17] [18] [19] .

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 と同じだが，SMS は起動する度に何度も送信される．
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れぞれのマルウェア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，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．その後，メソッドの先頭へログコードを挿入する．（これは実験 1 と同じ操作）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．

```
-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 6.1　getKeyNode

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．GoldDream

com.tecent.qqgame

iCalendar

iMatch

com.tencent.qq

3 個のマルウェアからは，ログを得ることはできたが，メソッド名が意味を為しておらず，挙動を特定することができなかった．beauty leg, breast, girl の 3 つ

ログが出なかったもの

6.3.2 実験 2 の結果

7　おわりに

おわりに

7.1　まとめ

7.2　今後の課題

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

[10] 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.

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

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

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

[14] 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.

[15] 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/.

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

[17] 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.

[18] 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.

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