Android の実機を利用した動的解析環境の提案

橋田 啓佑† 金井 文宏† 吉岡 克成† 松本 勉†
†横浜国立大学
240-8501 神奈川県横浜市保土ケ谷区常盤台 79-7
{hashida-keisuke-dk, kanei-fumihiro-tv}@ynu.jp
{ yoshioka, tsutomu }@ynu.ac.jp

あらまし 近年，モバイル向けOSであるAndroid が広く普及してきている．それに伴いAndroid OSを狙ったマルウェアはその数を増やすとともに高度化している．これに対して，Androidマルウェアの機能を把握する手段の1つとして，動的解析が広く行われている．しかしAndroidマルウェアの中には動的解析環境として利用されることの多いエミュレータやVM環境を検知して動的解析を逃れようとするものがあることから，今後，実機を利用した動的解析が重要になると考えられる．そこで本稿では動的解析における実機利用の利点と欠点を整理するとともに，Androidの実機を用いた解析を完全に自動で行う動的解析システムの実装例を示す．

Proposal of Malware Dynamic Analysis in Real Android Device

Keisuke Hashida† Fumihiro Kanei†
Katsunari Yoshioka† Tsutomu Matsumoto†
†Yokohama National University
79-7 Tokiwadai, Hodogaya-ku, Yokohama-shi, Kanagawa, 240-8501 Japan
{hashida-keisuke-dk, kanei-fumihiro-tv}@ynu.jp
{ yoshioka, tsutomu }@ynu.ac.jp

Abstract In recent years, devices with Android OS have become widespread. Malware targeting Android devices are increasing and becoming more sophisticated. As dynamic analysis of Android malware is widely conducted, some Android malware now have capability to escape the dynamic analysis by detecting VM environment which is often used as an analysis environment. Thus, dynamic malware analysis in real Android device is becoming more important. In this paper, we discuss advantages and disadvantages of real machine based dynamic malware analysis and show an implementation of the dynamic analysis system that can analyze Android malware in fully automatic manner.

1　はじめに

近年，スマートフォンやタブレット端末などのモバイル向け OS として Android が広く普及している．それに伴い Android OS を狙ったマルウェアはその数を増やすとともに高度化している．これに対してマルウェアの機能を明らかにすることで対策を行うため，様々な分析手法が検討されている．マルウェア分析の方法は静的解析と動的解析に大別されるが，本論文では動的解析を扱う．

Android マルウェアの動的解析の先行研究には論文[1][2]などがあげられる．これらの論文では動的解析環境は SDK 付属のエミュレータなどの仮想化環境を利用して構築されている．仮想化環境には同一の条件での解析の並列化の容易さや解析後における解析環境のクリーンナップの容易さなどの利点が存在する．その一方で，そうした仮想化環境を検知して動作を変化させることで動的解析を逃れようとするマルウェアも存在している[3]．さらに Android マルウェアには Bluetooth 通信などのハードウェアのエミュレーションが十分ではない機能を利用して感染拡大を狙ったものも存在している．こうした現状を踏まえて仮想化環境ではなく実機を利用した動的解析環境の重要性が今後高まると思われる．しかし実機を利用した解析環境に関する研究や考察は我々の知る限り非常に限定的であり，環境の構築方法も十分に公開されていない．

そこで本稿では Android における動的解析における実機利用の利点と構築の際の問題点について考察すると共に，Android の実機を用いた解析を完全に自動で行う動的解析システムの実装例を示す．さらに当該システムの有効性を検証するために，実機を利用した動的解析環境とエミュレータを利用した動的解析環境を用いて，同一のマルウェアの解析を行い，解析結果の差異を確認する．

本論文では，まず 2 章で Android マルウェアの動的解析に関する先行研究について説明する．3 章では実機を利用した解析環境における利点と欠点について述べる．

4 章では実機を利用した動的解析環境の構築方法を示す．5 章では提案システムとエミュレータを利用した動的解析環境での解析結果の比較を行い，その結果について述べる．最後に 6 章ではまとめと今後の課題について述べる．

2　関連研究

動的解析は，マルウェアの解析手法の一つである．解析環境内で実際にマルウェアを動作させその挙動を観測・分析することで解析を行う方法である．Android の動的解析の先行研究は多数存在するがここでは論文[1][2]について述べる．

論文[1]は Android における汚染追跡という情報フロー追跡技術を実現している．これは，端末内に保存されている重要情報にタグ付けを行い，情報をタグの伝播で追跡・監視をすることで情報の漏えいが行われているかをリアルタイムで監視し，判断することを可能としている．

論文[2]では仮想化技術を用いた Android マルウェア動的解析システムの提案，実装を行なっている．この手法においては，論文[1]において挙動の観測が難しかった，ネイティブコードの振る舞いについても観測が可能となっている．

上記のように Android の動的解析システムについて取り扱った研究は存在するが，これまでは仮想化環境にて構築を行うものがほとんどであり，実機上にシステムを構築する際の問題点や，構築法について論じた論文は少ない．

そこで本研究では実機上に動的解析環境を構築する際の利点と問題点についてまとめ，実際に解析環境を構築し有効性及び問題点の検証を行う．

3 実機を利用した動的解析環境の利点

動的解析の環境として実機を使う大きな利点として，仮想化環境の検知を行うAndroidマルウェアの解析があげられる．Androidマルウェアの中には，動的解析環境を検知するために実行している端末のモデル番号やメモリ上の文字列などから仮想化環境を検知し動作を変化させるマルウェアが登場している[3]．そうした仮想化環境の検知を行うマルウェアに対しても実機を利用した解析環境を用意することで解析を行うことができる．勿論，これらの仮想化環境の検知は，それらの検知を回避する仮想化環境の構築を行うことで仮想化環境での解析を行うことが可能であるが，実機を利用することでマルウェアと解析環境の改良競争を避けることができる．さらにマルウェアの中にはカメラ機能やBluetooth接続などのエミュレートの難しいハードウェア機能を利用したものが存在している．そうした検体に対して当該機能を保持している実機を利用して解析環境を構築することで解析を行うことができるという利点が存在する．

その一方で大量の検体に対して動的解析を行うことを想定した際に，仮想化環境では存在しなかった問題点として，解析後のクリーンナップがあげられる．通常仮想化環境を利用して動的解析を行う際は，解析前にクリーンな状態のスナップショットを作成し，解析後はスナップショットからイメージを作り直し，次の検体の解析に入る．しかしAndroidの実機を利用して動的解析を行う際はその方法が利用できない．そのため，クリーンナップには各パーティションをクリーンな状態に復元する必要がある．そのため，仮想化環境に比べて，一検体ごとにクリーンな状態に復元するための時間が必要となり，時間当たりの解析効率は落ちることになる．

4 提案システム

本章では第 1 節にて Android 端末のクリーンナップについて，パーティションの説明と利用したツールの説明を行い，第 2 節にて実機を利用した動的解析システムの実装例を示す．

4.1 クリーンナップについて

本節ではAndroidのパーティションの仕組みと動的解析の際に必要となるクリーンナップの仕組みについて説明を行う．

スマートフォンをはじめとした Android 端末のファームウェアは NAND フラッシュメモリ上をいくつかのパーティションに分け保存されている．パーティションは代表的なものに起動時のシステムコアとなる boot 領域やリカバリーモードで起動した際に使用されるファイルが収められた recovery 領域，通常起動で使用されるsystemファイルが収められているsystem領域，個人設定や追加したアプリなどが保存されるuserdata 領域などが存在する．これらのパーティションはパーティションごとにイメージファイルを利用して書換を行うことができる．その際にはroot権限を利用するか，ブートローダの状態で fastboot[4]を利用することで書き換えることが可能である．ただし，通常は公式に公開されているイメージファイル以外は利用できないようになっており，非公式のイメージファイルを利用するにはこの制限を解除する必要がある．これをブートローダアンロックという．

一般権限で動作するマルウェアは userdata 領域にインストールされ，インストール時に許可されたパーミッション内でしか動作することができないが，root 権限取得を行うマルウェアの中には system 領域に改変を加えるものや system 領域にマルウェアをインストールするものも知られている．こうしたマルウェアに対してクリーンナップを行うには単にインストールしたマルウェアをアンインストールするだけでは不十分である． root権限を取得したマルウェアによって加えられた変更を確実に排除し，クリーンな状態に戻すためにはすべてのパーティショ

ンをマルウェアの解析前の状態にもどす必要があるためである.

そのために各パーティションのクリーンな状態のイメージファイルのバックアップの作成を行っておき, 解析後にパーティションを上書きしレストアすることでクリーンナップを行う. バックアップの作成には Nandroid[5]を利用した. Nandroid は Android の NAND フラッシュメモリの各パーティションイメージをバックアップするために必要なツールとスクリプトのセットであり, Busybox[6]の導入及び root 取得の行われたリカバリーモード下で動作する. Busybox とは標準的な UNIX コマンドの機能を単一の実行ファイルで提供する組み込みシステム向けのアプリケーションである. Busybox を Android に導入することで Android の端末上で, 様々なコマンドを利用することが可能となる.

4.2 動的解析環境

提案システムは実際に検体が動作する犠牲端末と解析用マシン上で動作し, 解析処理全体を司る解析マネージャからなる.

・犠牲端末

Android マルウェアを実行する端末であり, USB ケーブルで解析用マシンと接続された実機である. 犠牲端末は あらかじめ非正規のイメージファイルを利用できるようにブートローダのアンロックを行い, 非正規のイメージを利用できるようにしておく. 犠牲端末には Nandroid をもとに環境に合わせて作成したバックアップおよびレストアを自動で行うスクリプトと必要なツールセットを導入したカスタムリカバリイメージを導入することで解析後のクリーンナップの自動化を行う.

・解析マネージャ

解析マネージャは動的解析の中心として, 解析者より与えられた条件をもとに検体のインストール, 実行, 各種解析の際のイベントの発生, 解析情報の取得, 解析後のクリーンナップを行う. 解析マネージャは adb[7], fastboot を利用して犠牲端末と USB ケーブルを介して操作を行い, 解析を進める. 検体のインストール・起動は Monkey[8]を利用した.

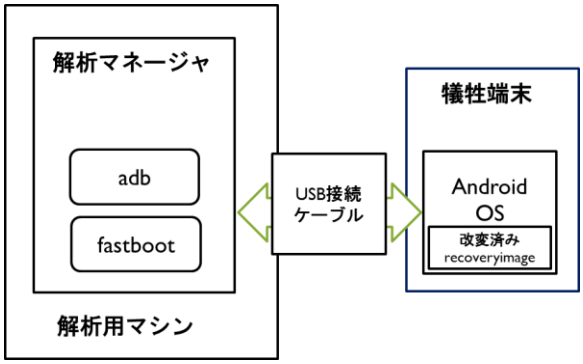

図 1 解析環境の構成図

犠牲端末として Samsung Google Nexus S を用いた. 搭載する AndroidOS としては, AndroidOS 4.0.3 をもとに recovery.img にのみ改変を施した OS を用いた. 犠牲端末からの通信はホストマシン上に設置したアクセスポイントを利用した Wi-Fi 通信でインターネット通信を行う. 解析マネージャはアクセスポイント上の通信データを tcpdump コマンドを使って取得すると共に, 犠牲端末の Logcat 出力を解析結果として取得する.

5 評価実験

本章では 4 章で説明した動的解析システムとエミュレータを利用した動的解析環境で, 同一のマルウェアの解析を行った際の解析結果の違いについて述べる.

5.1 実験目的

今回の実験は Android マルウェアの実機を利用した動的解析の有効性の検証として, 実機を利用した解析環境とエミュレータを利用した解析環境で同一の 100 検体について動的解析を行い, その差異の検証を行う. 更に, 実機を利用した動的解析のクリーンナップにかかる時間がどの程度必要になるかの調査を行った.

5.2 実験方法

実験に際して比較対象として, エミュレータを利用した解析環境の構築を行った. 具体的に

は，AndroidSDK 付属のエミュレータを利用して純正 OS のバージョン 4.0.3 を用いた動的解析環境を作成した．この環境はアクセスポイントを利用した通信の取得の代わりにエミュレータの機能としての tcpdump を利用した．解析後のクリーンナップ方法として起動直後の状態のスナップショットを事前に作成しておき，解析の際にはスナップショットから起動し，解析後はエミュレータを終了し新たにスナップショットから起動し直す方法を採った．

この 2 つの環境で Android マルウェア 100 検体について 5 分間の解析を行った．検体の検知名の一覧は下の表のとおりである．検体名は Trend Micro による検知名，検体数は実験に使用した検体の数を示している．ただし Android.Enesoluty と Android.Dendoroid については，実験に利用した検体が Trend Micro では検知されなかったため，Symantec における検知名を記述している．

表 1 実験にしようしたマルウェア検体一覧

検知名	数	検知名	数
AndroidOS_ADRD.BLK	1	AndroidOS_FAKEINST.BLK	1
AndroidOS_ADWAirpush.A	3	AndroidOS_FAKENOTIFY.A	2
AndroidOS_ADWAIRPUSH.BLK	1	AndroidOS_GAPPUSIN.BLK	1
AndroidOS_AGENTBLK.359	1	AndroidOS_GINGERMASTER.A	1
AndroidOS_AGENTBLK.367	1	AndroidOS_JIFAKE.CL	1
AndroidOS_AGENTBLK.369	1	AndroidOS_KMIN.B	2
AndroidOS_AGENTBLK.772	1	AndroidOS_KMIN.SMA	1
AndroidOS_ANSERV	1	AndroidOS_KUNGFU.EPJ	1
ANDROIDOS_BASEBRIDGE.CTE	2	ANDROIDOS_KUNGFU.VTD	2
AndroidOS_BOTPANDA.A	2	AndroidOS_LOTOOR.A	1
AndroidOS_BOXER.DA.A	2	AndroidOS_LOTOOR.AA	1
AndroidOS_BOXER.DY	3	AndroidOS_MOGHAVA.A	1
AndroidOS_BOXER.EA	5	AndroidOS_Nickspy.D	1
AndroidOS_BOXER.EMS	1	AndroidOS_OPFAKE.CDC	1
AndroidOS_BOXER.MC	1	AndroidOS_OPFAKE.EJL	2
AndroidOS_BRIDGE.AO	1	AndroidOS_PLANKTON.BLK	2
AndroidOS_BRIDGE.AW	1	AndroidOS_Plankton.D	1
AndroidOS_BRIDGE.B	1	AndroidOS_RUFRAUD.AZ	1
AndroidOS_BRIDGE.M	1	AndroidOS_SMSAGENT.AB	2
AndroidOS_DIALER.A	1	AndroidOS_SMSAGENT.AD	1
AndroidOS_DOUGALEK	1	AndroidOS_SMSAGENT.AD	2
AndroidOS_DROIDKUNGFU.A	1	AndroidOS_SMSAGENT.BLK	1
AndroidOS_ENHANGS.A	1	AndroidOS_SMSHIDER.CDC	1
AndroidOS_FAKE.AL	1	AndroidOS_STEEK.A	1
AndroidOS_FAKE.AR	1	AndroidOS_THEFTAWARE.CDE	1
AndroidOS_FAKE.B	2	AndroidOS_TRACK.Z	1
AndroidOS_FAKE.BLK	5	AndroidOS_TROJFAKEInst.A	1
AndroidOS_FAKE.BT	1	AndroidOS_TROJOpfake.F	1
AndroidOS_FAKE.CDE	1	AndroidOS_TROJPlankton.A	2
AndroidOS_FAKE.CW	1	AndroidOS_TROJSPYCosha.A	1
AndroidOS_FAKE.DQ	10	Android.Enesoluty	1
AndroidOS_FAKE.MC	4	Android.Dendoroid	1
AndroidOS_FAKEBROWS.A	1		

5.3 実験結果

5 分間の解析の結果，実機を利用した解析環境ではマルウェアに由来する通信以外にもプレインストールされたアプリケーションや OS が行う通信が大量に行われていることが分かった．そこでマルウェアを動作させない状態で 30 分間の通信を行い，ホワイトリストの作成を行い，それらの通信をできる限り取り除くことでマルウェアに由来する通信の抽出を行った．100 検体のうち正常にインストールすることに成功したものは実機・エミュレータともに 95 検体だった．そのうちで Logcat 出力より起動に成功していることが確認できるものはエミュレータが 88 検体，実機が 90 検体だった．通信を行った検体はエミュレータが 30 検体，実機が 31 検体だった．また 100 検体のうち 5 検体について，エミュレータ上では発生しない接続先への名前解決を実機上では試みることが分かった．これらの検体については仮想化環境の検知を行い，振る舞いを変化させている可能性がある．そこでこれらの検体については動的解析後に静的解析を試みた．名前解決を試みるドメインの総数はエミュレータが 46 個に対して実機は 51 個であった．解析にかかった時間は実機を使った解析環境では 12 時間 15 分に対してエミュレータをつかった解析環境では 8 時間 30 分であった．実機におけるクリーンナップの時間は一検体あたり約 2 分 30 秒であり，予想通り解析にかかる時間的コストが大きくなる結果となった．

表 2 動的解析結果

	エミュレータ	実機
インストール成功数	94 検体	94 検体
起動に成功	88 検体	90 検体
通信を行った検体数	30 検体	31 検体
名前解決を行ったドメイン数	46 個	51 個
総解析時間	8h30m22s	12h15m39s

実機でのみ通信を行った 5 検体について静的解析を行った結果，そのうちの 4 検体については検体が実行環境の情報を取得し，動作を変化させていることが分かった．

たとえば，今回の実験で使った Android.Dendoroid ファミリに属する検体(ハッシュ値： 099a57328de9335c524f44514e225d50731c808145221affdd684d8b4dad5a1d) の仮想化環境を検知している部分が図 2 である．

この部分では SDK 付属のエミュレータを検知するために動作中の端末のビルド情報をエミュレータ特有のビルド情報と比較することでエミュレータの検知を行い，振る舞いを変化させていることが確認できる．

ただし，広告モジュールにおいてもこうした検知を行っており，今回発生した通信の中にも広告が入っているものと思われる．

6 考察

100 検体を解析した結果，実機とエミュレータにおいて明確に通信の発生に差異が出るものが存在することを確認した．一方で，予想された通り，時間的コストにおいてはエミュレータに比べて実機は大きくなってしまうことが確認でき，その差は今回の 5 分間の短期の解析においてきわめて大きな割合を占めることとなり，解析の効率を大きく落とすこととなった．

また 100 検体の中には実機・エミュレータを問わず，起動はしても通信を発生させないものが多く含まれていた．これらの検体の Logcat 出力を確認したところ，SMS の送信を試みているものが存在した．さらにそれらの検体は TCP/UDP 通信ではなくプレミアム SMS を利用した課金を行うマルウェアファミリーに属しており，ほとんど通信を発生させていないことが分かった．このことから，WiFi を介した通信のみの環境では，上記のようにモバイル通信を利用する検体の動的解析には不十分であることがわかった．

今回の環境ではモバイル通信に接続していない実機を使って動的解析を行ったため，SMS を利用した挙動は観測できなかったが，理論上はそうした挙動もモバイル通信に接続した実機上で動作させることで観測が可能になるはずである．その一方で，現在の WiFi を利用した通信以上に注意深い通信の制御が必要となるため，観測の方法について検討していく必要がある．

7 まとめと今後の課題

本稿では Android における実機を利用した動

```
this.threadPre = new Thread() {
    public void run() {
        long v8 = 60;
        Looper.prepare();
        if(MyService.this.GPlayBypass.booleanValue()) {
            while(true) {
                if(!PreferenceManager.getDefaultSharedPreferences(MyService.this.getApplicationContext())
                        .getBoolean("Start", false)) {
                    System.currentTimeMillis();
                    if(!"google_sdk".equals(Build.PRODUCT) && !"google_sdk".equals(Build.MODEL)
                            && !Build.BRAND.startsWith("generic") && !Build.DEVICE.startsWith(
                            "generic") && !"goldfish".equals(Build.HARDWARE)) {
                        PreferenceManager.getDefaultSharedPreferences(MyService.this.getApplicationContext())
                                .edit().putBoolean("Start", true);
                        MyService.this.initiate();
                    }
                }
                else if(PreferenceManager.getDefaultSharedPreferences(MyService.this.getApplicationContext())
                        .getBoolean("Start", false)) {
                    MyService.this.initiate();
                }

                try {
                    Thread.sleep(MyService.this.interval);
                }
                catch(Exception v0) {
                    MyService.this.threadPre.start();
                }
            }
        }

        MyService.this.initiate();
    }
};
```

図 2 エミュレータの検知コード

的解析システムの構築についての提案を行い，実際に構築した実機を使った動的環境の有効性と課題についてエミュレータを利用した解析環境と比較を行い，検証を行った．Android マルウェアの中にはエミュレータ上では動作しないが実機上では動作するものが存在するため，そうしたマルウェアが今後さらに増えれば実機を利用して動的解析を行う意義は大きくなっていくと考えられる．その一方で，単純に WiFi 上の通信を監視するだけでは，挙動の観測としては不十分なことも分かった．SMS を利用したマルウェアの動的解析にはモバイル通信に接続が必要であると考えられる．

今後の課題としては，実機を使うメリットを大きくするためには単純な通信を監視するだけではなく様々なマルウェアの挙動を監視できるように環境を構築していく必要がある．

参考文献

[1] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick Mc-Daniel, and Anmol N. Sheth. , “TaintDroid: An Information-Flow Tracking System for RealtimePrivacy Monitoring on Smartphones,” In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October, 2010. Vancouver, BC.

[2]Lok Kwong Yan, Heng Yin “DroidScope：Seamlessly Reconstructiong the OS nad Dalvik Semantic Views for Dynamic Android Malware Analysis” USENIX security 2012.

[3] “Android OBAD Technical Analysis Paper Comodo Malware Analysis Team July 20013” http://www.comodo.com/resources/Android_OBAD_Tech_Reportv3.pdf, last visited 2014/08/23.

[4] “Building for devices Android Developers”https://source.android.com/source/building-devices.html, last visited 2014/08/23.

[5] “Nandroid v2.0 - Full NAND backup and restore tool (tried and tested!)”, http://forum.xda-developers.com/showpost.php?p=3046976, last visited 2014/08/23.

[6] “BusyBox” http://www.busybox.net/, last visited 2014/08/23.

[7] “Android Debug Bridge Android Developers” http://developer.android.com/tools/help/adb.html, last visited 2014/08/23.

[8] “UI/Application Exerciser Monkey Android Developers” http://developer.android.com/tools/help/monkey.html, last visited 2014/08/23.