修 士 論 文

FireMarking:Android アプリケーションの セキュリティ指向サジェスションシステム

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

加藤 邦章

2015 年 3 月

修 士 論 文

FireMarking:Android アプリケーションの セキュリティ指向サジェスションシステム

指導教員　篠田陽一 教授

審査委員主査　篠田陽一 教授
審査委員　丹康雄 教授
審査委員　知念賢一 特任准教授

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

1210017 加藤 邦章

提出年月: 2015 年 2 月

概 要

Android ユーザから Android マルウェアが原因とされる被害報告は年々増えている。G Data 社のレポートでは、ユーザにとって迷惑な行為をするアプリケーションによる被害が増加していると述べている。McAfee 社のレポートでは、正規のアプリケーションやサービスの脆弱性を悪用して、アプリケーションストアやセキュリティ製品の監視網を欺くマルウェアが増え、現在のマーケットのセキュリティレベルでは脅威を阻止できないと警告している。その他、シマンテック社のレポートでは、マルウェアによる被害拡大は、ユーザの意識に問題があると説明している。

この 3 つのレポートから、 Android マルウェアによる被害の拡大の原因は、次の 2 つであると考える。それは、ユーザは十分な情報に基づいた意思決定によって、アプリケーションをインストールしているわけではないことと、Android マーケットはマーケット運営者によって、適切に管理されているわけではないことである。本研究では、これらの問題を解決するため、第 3 者の視点で Android マーケットを分析して、マーケットや公開されているアプリケーションの利用リスクを示すセキュリティ指向サジェスションシステム、FireMarker を提案する。

前述の 2 つの問題と Android セキュリティに関する既存研究から、FireMarker の利用者を Android アプリケーション操作により生じる情報を欲する Binarian、Android マーケットの傾向分析を行う Analyst、マーケットやアプリケーションの有用な情報を得たい End-user に分けた。彼らの要求を推測し、それを満たすための条件を議論した。その結果、FireMarker には、次の 3 つの条件があると考えられる。1 つ目の条件は、短時間で大量のアプリケーションから情報を集めるために、複数のマシンや Android 端末を制御できること、2 つ目の条件は、アプリケーションを操作した時に生じる情報を取得するために、アプリケーション操作の自動化が可能なこと、3 つ目の条件は、ユーザにマーケットやアプリケーションの利用リスクを示すために、情報を解析、評価して、結果を可視化することである。これらの条件を満たすため、FireMarker は Android マーケットを調査し、その傾向を数値化する MarketDrone と、集めた情報を利用して、ユーザにマーケット及びアプリケーションの利用リスクを示す FireMarking の 2 つから構成されるシステムとして設計した。

MarketDrone は各マーケットからアプリケーションをダウンロードする Crawler、複数のマシン、複数の Android 端末を利用して、アプリケーション操作時に生じるシステムログやトラフィックを収集する Dispatcher、その結果を基にマーケットの傾向を数値化する Filter、前述の 3 つのモジュールを統合し、自動化する Controller から構成される。そして、FireMarking は MarketDrone が出力した情報を用いて、マーケット及びアプリケーションの利用リスクを測る。

本研究では FireMarker の設計のうち、MarketDrone の Crawler、Dispatcher、Filter を次のように実装した。Crawler は、APKTOP という Android マーケットをクロールするツールとして実装した。Dispatcher は、情報収集の処理速度を追求した設計パターン A と可用性を重視した設計パター

ン B を考案し実装した。Filter は、アプリケーションのシステムログを用いて強制終了や管理者権限の要求するアプリケーションを特定する。また、トラフィックからは、FQDN を種類別に分ける。さらに、トラフィックから HTTP 通信を抽出し、広告に関する通信を行うアプリケーションを特定するツールとして実装した。

次にこれらを用いて、APKTOP と既存のクローラを用いてアプリケーションを集めた OperaMobileStore を調査する実験を行った。その結果、APKTOP からはアドウェアを持つ 4 つのアプリケーションを発見し、OperaMobileStore からはアドウェアを持つ 8 つのアプリケーションを発見した。さらに、両方のマーケットで日本で行った実験にも関わらず、エストニアやロシア、中国など日本以外の広告を表示するアプリケーションを発見した。その他にも、APKTOP からトロイの木馬であるアプリケーションを特定した。実験結果から FireMarker は、APKTOP と OperaMobileStore を比べたとき、ユーザにとって安全なマーケットは OperaMobileStore であると示した。

最後に、実験によって明らかになった課題を今後の展開としてまとめ、その解決方法について議論した。その議論において、実験結果から各アプリケーションの利用リスクを評価する手法、マーケットの利用リスクを測る方法を提案した。FireMarker によって、Binarian は提案する手法を大量のアプリケーションを使って、検証することができる。また、調査したマーケットの情報をデータベースに保存しアクセスできるようにすれば、実験のデータセットとして利用することができる。さらに、Analyst は各マーケットごとの情報を入手し、強制終了するアプリケーションや管理者権限を要求するアプリケーションの数、FQDN の種類、迷惑な広告を表示させるアプリケーション数などの情報を入手することができる。その結果、彼らはマーケットの動向を容易に調べることができる。さらに、End-user は FireMarker によって、アプリケーションインストール時に利用するマーケットの安全性を確かめることができる。その結果、マーケットの安全性を考慮した意思決定により、アプリケーションを使うか否か決めることができる。以上をもって、本研究は Android マルウェア被害拡大の原因とする 2 つの問題を解決した。

目 次

図 目 次

表目次

第1章　序論

スマートフォンの普及に伴い、多くの人がインターネットを利用する機会が増えた。特に Android が搭載された端末（スマートフォンやタブレット PC など以下、Android デバイスと略記する。また Android が搭載されたエミュレータを Android エミュレータと呼ぶこととする）が急速に普及している。

Android[1] とは Google 社を中心として組織された規格団体 Open Handset Alliance（OHA）が発表した、OS、ミドルウェア、そして主要なアプリケーションを含めモバイルデバイス開発に適切なソフトウェアスタックを提供するフレームワークである。このフレームワークはオープンソースであり、誰もが無償で入手することができる。また、独自の拡張を加えることもできる。AndroidOS は携帯端末用にカスタマイズされた Linux カーネルを実装しており、GNU/Linux や CentOS といった Linux ディストリビューションと異なる点は、Android は内部構造に Dalvick 仮想マシン（OS バージョン 4.3 以上は ART 仮想マシン）というサンドボックスを持ち、全ての Android アプリケーションはその上で作動することと、パーミッション機構を採用していることである。パーミッション機構とは、アプリケーションが利用する機能や情報をインストール時にユーザに示して承認を求める仕組みである。

アプリケーションはサンドボックスの上で実行され、付与されたパーミッションに応じて、保護された機能や情報へアクセスを行う。そのため、WindowsOS を搭載した端末に見られる悪意あるアプリケーション（以下、マルウェアと呼ぶ）の自動感染の可能性は低い。ユーザがアプリケーションのパーミッション機構を理解していれば、ある程度マルウェアの脅威を阻止できる。

それにも関わらず、Android マルウェアは増加の一途を辿っている。2014 年 4 月に報告された G Data 社の Mobile Malware Report[2] では、2013 年から 2014 年の間に約 120 万もの新たな Android マルウェアを確認し、2012 年の増加数と比べると 4.6 倍に達したと報告している。これは約 3,600 件の新種が毎日登場した計算になる。

Android マルウェアはプログラムの特性に基づいて、2 つに分類することができる。それは明確に不正プログラムと認められるものと Potentially Unwanted Program（以下、PUP と略記する）という不正ではないが、ユーザにとって迷惑な行為をするプログラムである。PUP は例えば、ユーザが望んでいない広告を勝手に表示させる。また、Web へのアクセス履歴を収集して、外部サーバに送信する、といった動作を行う。PUP の注目すべき点は、ユーザが許可をして、ユーザの Android デバイスに導入されているため、不正プログラムか普通のプログラムか、定義が非常に曖昧であ

る点と、一度導入されると、通常の方法では完全な削除が困難である点である。この PUP に分類されるアプリケーションが近年激増しており、被害が多数報告されている。

その他にも、2014 年 6 月に報告された McAfee Labs Threats Report[3] によれば、Android マルウェアの多くは、Android プラットフォーム標準の API を利用して、重要な情報を盗み出すことや、高額料金の SMS を送信といったことを行っている。しかし、最近は Android プラットフォームが提供する機能を悪用するだけではなく、アプリケーションストアやセキュリティ製品の監視から逃れるために、正規のアプリケーションやサービスの脆弱性を悪用するケースが増えてきていると報告している。そのため、Android マーケット、Android デバイス向けデジタルコンテンツ（アプリケーション、映画、音楽、書籍など）を配信するサービスの現在のセキュリティレベルでは脅威を阻止できないと示唆している。

このような Android マルウェアの感染、被害拡大はユーザ意識にも問題があると、シマンテック社のノートンモバイルアプリ調査 [4] は報告している。多くの人は所有している Android デバイスへのウイルス感染の懸念はしているが、リスクに対する情報に疎く、Android アプリケーションをインストール際に、何に同意したか理解していないと述べている。

この 3 つのレポートから、マルウェア被害の拡大の原因は次の 2 つであると考える。1 つ目は、ユーザは十分な情報に基づいた意思決定を行って、アプリケーションをインストールしているわけではないこと、2 つ目は、Android マーケットはマーケット運営者によって、適切に管理されているわけではないため、ユーザは利用するマーケットの安全性を知らずに利用している可能性がある。これらの問題を解決するため、本研究では、第 3 者の視点で Android マーケットを分析して、マーケットや公開されているアプリケーションの利用リスクを示すセキュリティ指向サジェスションシステム、FireMarker を提案する。

FireMarker は次のことを行う。

- 複数の Android マーケットをクロール、アプリケーションをダウンロードする。
- Android エミュレータ/デバイスを利用して、アプリケーションの操作を自動的に行い、情報を収集する。
- 出力された結果を基に Android マーケットの傾向を数値化する。
- この数値化された傾向を用いて、Android マーケット及びアプリケーションの利用リスクを測る。

FireMarker によって、ユーザは導入するアプリケーションがどのマーケットに属し、マーケットの利用リスクを理解することで、疑わしいアプリケーション導入を避ける判断の一助となると考える。

まず次章では、Android プラットフォームと既存研究について紹介する。３章では、FireMarker の利用者の要求を定義し、FireMarker の実装に必要な条件を説明する。4 章では、FireMarker の設計と実装を述べ、5 章でこれを使った実験の結果と考察を述べる。最後に 6 章で、実験によって明らかになった今後の仕事と本研究の結論を述べる。

t

第2章　Androidとそのセキュリティに関する既存研究

本章では、まず初めにAndroidプラットフォームを説明する。ここではFireMarkerと強く関係がある部分について詳しく述べる。次にAndroidモバイルセキュリティに関する既存研究を紹介する。

2.1　Androidプラットフォーム

2.1.1　Androidとは

Android[1]とはGoogle社を中心として組織された規格団体Open Handset Alliance（OHA）が発表したオープンソースプラットフォームである。Androidを構成するコンポーネントのほとんどは、Apache v2ライセンスで提供されており、ほぼ無償で利用できる。また、Androidは特定のハードウェアに依存せず、Androidが搭載されているデバイスならばアプリケーションは作動する。Androidのアーキテクチャは次の5つのコンポーネントから成り立つ。

Linuxカーネル

Androidは情報携帯端末用にカスタムされたLinuxカーネルを基盤としている。このコンポーネントは、メモリー管理や電源管理、ハードウェアの管理を行う。

標準ライブラリ

標準ライブラリはLinuxカーネル上で動作するネイティブライブラリである。ハードウェアそのものを制御するライブラリやデータベース機能を提供するSQLite、Webブラウザの描画を管理するWebKit等が含まれている。

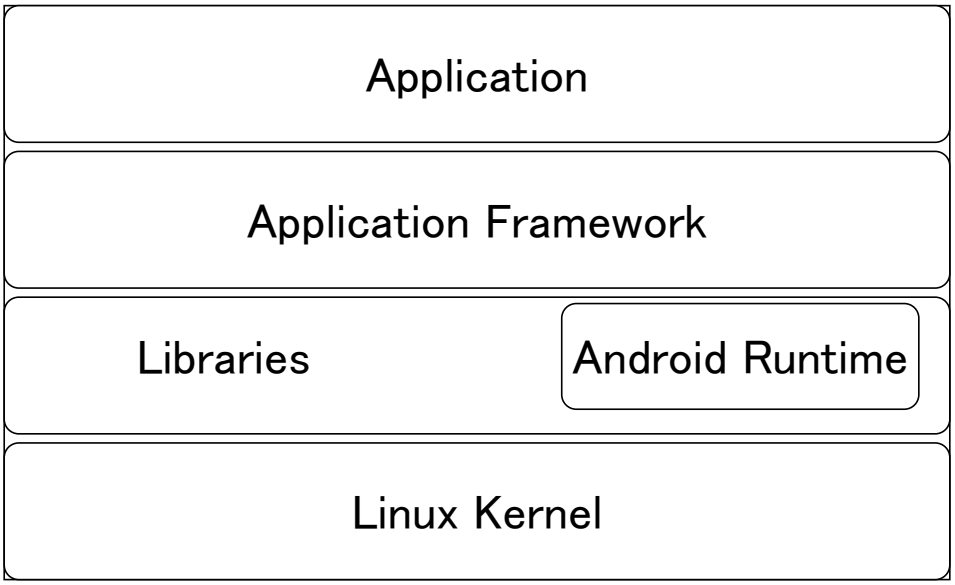

図 2.1: Android のアーキテクチャ

Android ランタイム

Android ランタイムは Android アプリケーションを実行するための環境を提供している。これには、Dalvik 仮想マシン（OS バージョンが 4.3 以上は Android ラインタイム仮想マシンと呼ぶ）と Java のコアライブラリで構成されている。Dalvik 仮想マシンは携帯端末の低メモリー環境に最適化されたサンドボックスで、Android アプリケーション内の Java のバイトコードを Dalvik バイトコード（dex）と呼ばれる中間言語に変換して実行している。

アプリケーションフレームワーク

アプリケーションフレームワークはアプリケーション開発に必要な機能を提供する。また、ユーザインターフェースの表示やデータ共有、位置情報、通話、通知などを管理する。

アプリケーション

ブラウザや電話、カメラ等、エンドユーザが利用するアプリケーションが属するコンポーネントである。標準で提供されているアプリケーションに加え、Android マーケットの GooglePlay[5] などで提供されている有償/無償アプリケーションや開発したアプリケーションを導入することができる。

2.1.2 パーミッション機構

パーミッション機構 [6] とは、アプリケーションが利用する機能や情報をインストール時にユーザに示し、ユーザからアクセスを許されたときのみ、アプリケーションをインストールできる仕組みである。例えば、図 2.2 は Android 用のカメラアプリケーションをインストールごとに同意するボタンを押す前に表示される画面である。このカメラアプリケーションは、ユーザに端末上のアカウント、連絡先、位置情報などのアクセスを要求していることがわかる。ユーザがこの要求に同意し、同意ボタンを押せばアプリケーションのインストールが始まり、アプリケーションに要求された情報へのアクセスに必要なパーミッションが付与される。逆に同意せず、この画面から別画面に移れば、アプリケーションのインストールは始まらない。この仕組みと Dalvik 仮想マシンによって、アプリケーションは仮想マシンの上で実行され、付与されたパーミッションに応じて、保護された機能や情報へアクセスを行う。

パーミッションの項目はアプリケーションファイル（以下、APK ファイルと呼ぶ）のルートディレクトリにある AndroidManifest.xml と呼ばれるファイル（以下、マニフェストファイルと呼ぶ）に記述されている。この APK ファイルは、Java アプリケーションの配布に用いられる JAR 形式と似た形式で、Zip 形式で圧縮された書庫ファイルとなっており、内部には、マニフェストや Java のライブラリが格納された class.dex をはじめ、規定の形式で命名・配置されたファイルやフォルダが格納されている。その他にもマニフェストファイルは次のことが記述されている。

- アプリケーションのパッケージ名
- アプリケーションのコンポーネント、クラスの起動条件
- アプリケーションのスタート画面となるコンポーネント名（以下、メインアクティビティと呼ぶ）
- アプリケーションが必要とする最低限の Android API
- アプリケーションが必要とするライブラリ

Android アプリケーション開発のツール

Android アプリケーション開発には Android SDK（Software Development Kit）が無償提供されている。Android SDK によって、Android デバイスとマシンが USB を介して接続し、アプリケーションをデバイスもしくはエミュレータ上で実行しながらデバッグすることができる。そのデバッグツールを Android Debugger Bridge（以下、adb と呼ぶ）[7] と言う。adb は主にエミュレータのインスタンスや Android 搭載のデバイスの状態の管理する adb client、adb server、adb deamon（以下

図 2.2: Android アプリケーションのインストール時の確認画面

adbd とする）の 3 コンポーネントからなる adb client/adb server・プログラムである。各コンポーネントの役割は次のとおりである。

- adb client
  ローカルホストで動き、adb shell や adb logcat などのコマンドを呼び出し、adb server を介してターゲットの adbd と通信する。adb client は複数起動することができる。

- adb server
  ローカルホストのバックグラウンドで動作し、adb client と adbd の間で通信の管理を行う。adb server は 1 ホストにつき、1 つまでしか起動できない。

- adbd
  Android 搭載型端末及びエミュレータのバックグラウンドで動作し、adb client と通信を行う。adb client からの命令を端末本体のプロセスで実行する役割を持つ。

adb client を開始すると、adb client はまず adb server プロセスがすでに起動されているか確認をする。起動していない場合は adb server プロセスを開始する。adb server が開始すると、ローカルの TCP ポート 5037 をバインド、adb client から送られるコマンドを監視する （すべての adb client はポート 5037 で adb server と通信する）。

次に adb server は、実行中の全ての Android エミュレータ/デバイスのインスタンスとの接続をセットアップする。adb server は、5555 から 5585 の偶数番号のポート、およびエミュレータ/デバ

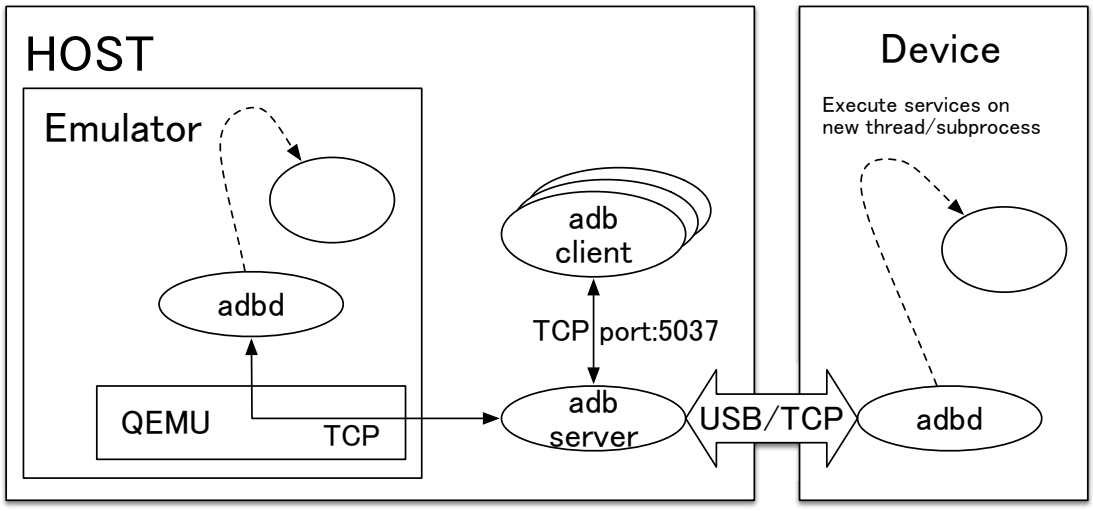

図 2.3: adb の全体構成

イスで使用されるポートをスキャン、エミュレータ/デバイスのインスタンスを調べる。adbd が見つかると、そのポートへの接続をセットアップする。各エミュレータ/デバイスのインスタンスは連続するポートのペア（コンソール接続用の偶数番号のポートと adb 接続用の奇数番号のポート）を利用する。すべてのエミュレータ/デバイスのインスタンスへの接続がセットアップできれば、コマンドを使用してインスタンスを制御およびアクセスできる。最終的に adb は図 2.3 のように Android エミュレータ/デバイスと接続して、制御する。

例えば、adb client から adb shell ls コマンドが送信された場合、図 2.4 のように処理される。まず、adb client が adb server に向かって、”host:version”と送信する。そして、バージョン番号を確認し、次に”host:transport:deviceID”を送り、送信先を切り替える。最後に”shell:ls”が Android エミュレータ/デバイス側の adbd に転送され、実行される。実行による標準出力は adb server、adb client を介して、標準出力としてコンソールに出力される。

2.2 Android セキュリティに関する既存研究

2.2 節では、FireMarker に関係する既存研究を Android マーケット及び Android の内部構造のどのコンポーネントを対象とした研究か、分けて紹介する。

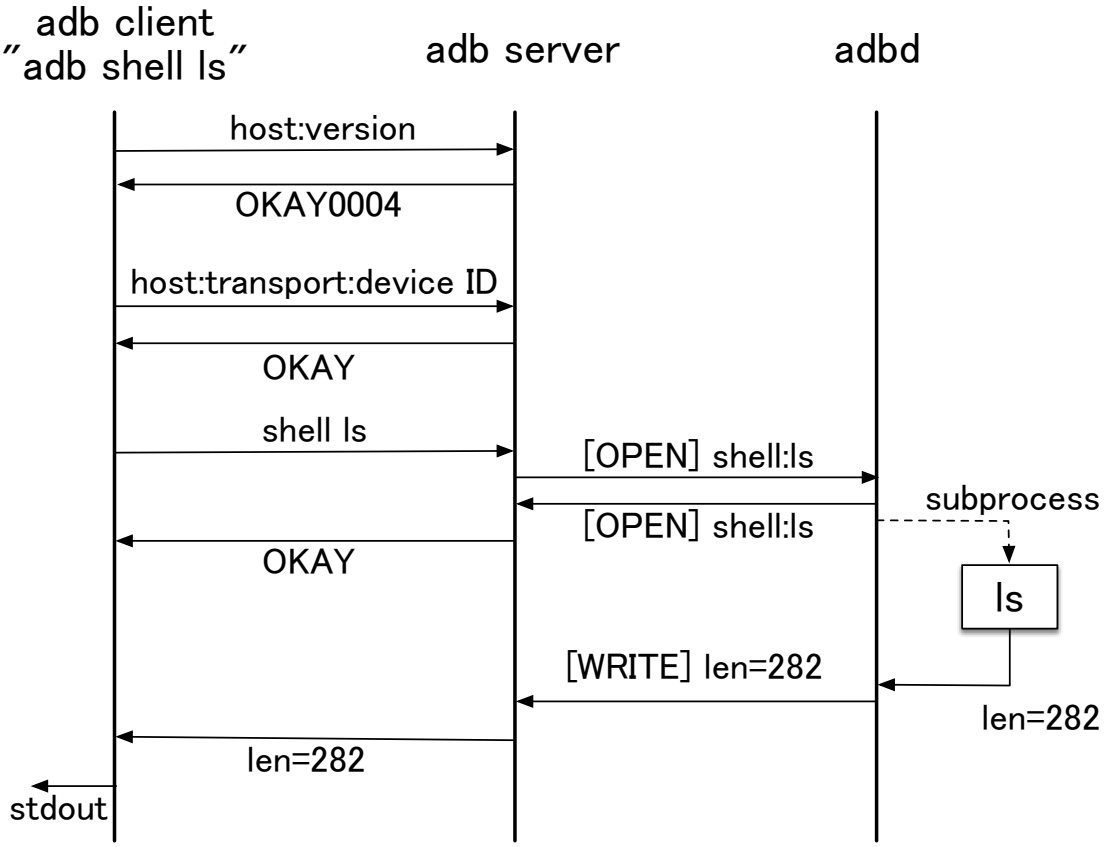

図 2.4: adb shell ls のシーケンス図

2.2.1 Android マーケットに焦点を当てた研究

Nicolas ら [8] は、GooglePlay を調査するため、最初の大規模なクローラである PlayDrone を開発した。これは様々なハッキング技術を利用して、アプリケーションをダウンロードするクローラとダウンロードしたアプリケーションを逆コンパイルすることで得られるソースコードを解析する機能を持つ。5ヶ月に渡る観測の結果、GooglePlay に登録されるアプリケーションの月ごとの増減率やダウンロード回数との関係性、無料と有料アプリケーションの平均登録数及び平均ダウンロード数と人気アプリケーションの相関性を明らかにした。さらに、コード解析の結果、多くのアプリケーションに OAuth の秘密鍵が埋め込まれていることを明らかにした。

Yajin ら [9] は、Android マーケットで公開されている悪質なアプリケーションを系統的に検知するため、パーミッションベースの行動フットプリンティングとヒューリスティックベースのフィルタリングの 2 方式が実装された検知ツール、DroidRanger を開発した。これを用いて、2011 年 5 月から 6 月の間に 5 つの Android マーケットから集めた 204,040 アプリケーションを分析したところ、211 ものマルウェアを発見した。そのうち 2 アプリケーションがゼロデイ攻撃を有すると明

らかにした。

2.2.2 アプリケーションコンポーネントに焦点を当てた研究

Wei ら [10] は、Android アプリケーションで要求される権限の利用の内容を分析するため、Permlyzer を開発した。これは従来のアプローチとは異なり、アプリケーションがどの場面で、どんな目的で権限を利用するのか、様々な側面から正確に権限の利用情報を得るため、コールスタックベースの解析手法を提案している。Permlyzer を用いた実験では 113,237 の無料アプリケーションと 51 マルウェア/スパイウェア・ファミリから、権限利用の特性を明らかにした。その結果、マルウェア/スパイウェア・アプリケーションはユーザインターフェースを介さず、アプリケーションのホーム画面であるメインアクティビティで危険な権限の要求を多数行っていることを明らかにした。

Jonathan ら [11] は、エミュレータ上で多くのアプリケーションを自動的に実行させ、悪質な広告を表示させる分析ツール、MAd Fraud を開発した。これは、アプリケーションがバックグラウンドにあるときに、広告を要求したり、ユーザが広告をクリックしていないにも関わらず、クリックしたと偽装する 2 つの詐欺的な振る舞いを特定することができる。GooglePlay 及びサードパーティマーケットを含む 19 の Android マーケットから集めた 130,339 アプリケーションとセキュリティ企業によって提供されている 35,087 アプリケーションの 2 つのデータセットを MAd Fraud に適用したところ、約 30%のアプリケーションがバックグラウンドで広告を要求していることを明らかにした。加えて、クリックしたことを偽装するアプリケーションを 27 も発見した。

磯原ら [12] は、正規アプリケーションに悪意ある Android アプリケーションが内包されたマルウェアをセカンドアプリケーションと定義し、これを検知する手法を提案した。この検知手法は、Android アプリケーション内部に ZIP 形式のファイルを探し、Android アプリケーションの固有ファイルの有無の判定（マニフェストファイル及び class.dex の有無の判定）を行い、2 つのファイルを発見した場合をセカンドアプリケーションと判定する、というものである。Android Market（現：GooglePlay）と一般 Web サイト [13] から収集した合計 600 のアプリケーションについて検査を行った。そして 14 アプリケーションがマルウェアと判定され、うち、13 アプリケーションがエクスプロイトコードとセカンドアプリケーションを含むため、明確にセカンドアプリケーション内包型 Android マルウェアと判定できた。この実験によって、提案手法は、アプリケーションの事前審査等に用いることで、未知のマルウェアを検知できることを示した。

2.2.3 Linux カーネルコンポーネントに焦点を当てた研究

William ら [14] は、個人情報の追跡ができる Taint 解析を利用したシステム、TaintDroid を開発した。Taint 解析とは、source と呼ばれる情報に Taint という識別子を付与し、その情報が使用されたとき、使用された情報と共に伝搬する Taint の流れを解析する手法である。このシステムは

Linux カーネルに独自のコードを加えることで、利用可能である。実験によれば、AndroidMarket（現:GooglePlay）からランダムに選んだ 30 アプリケーションのうち、3 アプリケーションが個人情報を、15 アプリケーションが位置情報を外部サーバに送信していることを明らかにした。

Yan ら [15] は、仮想化技術による Android マルウェアの解析プラットフォーム、DroidScope を開発した。これは AndroidOS に独自 API を組み込み、OS と Java コードレベルの解析を行うことができる。また、ネイティブな Dlavik 命令トレースや、プロフィール API レベルのアクティビティを収集し、Taint 解析を利用して、Java とネイティブコンポーネントを通した情報漏洩を追跡するツールを開発した。実験において、Android マルウェアである DroidKungFu と DroidDream を解析したところ、マルウェアの挙動だけでなく、Java とネイティブコンポーネントの相互作用による挙動を明らかにした。

葛野 [16] は、ネットワーク通信と端末上の情報にアクセス可能なアプリケーションによって、ユーザが意図しない情報の外部送信が起こる問題を解決するため、アプリケーションの通信を監視し、パーミッションの組み合わせによって発生する脅威を防ぐ手法を提案した。提案手法を実現するために開発された情報フロー制御アプリケーションとカスタマイズされた OS を用いて、HTTP パケットを監視、個人情報や位置情報などの機密情報に該当する情報を検知、制御する。提案手法の評価実験から、監視対象アプリケーションは端末情報や位置情報の送信していることを明らかにした。

第3章　FireMarkerの要件定義

本章では前章を踏まえ、現在のAndroidセキュリティと2.2節で紹介した既存研究から考えうる課題を説明する。そして、FireMarkerを利用するユーザを定義、彼らの要求を推測し、その要求を満たす条件について議論する。

3.1　Androidセキュリティの課題

2.1節で、Androidにはパーミッション機構という仕組みが備わっていることを説明した。しかし、ユーザは何に同意してアプリケーションをインストールしたのか理解しておらず、パーミッション機構を適切に運用できていない。さらに、正規のサービスやアプリケーションを利用したAndroidマルウェアが増えている状況下で、アプリケーションインストールの判断に必要な情報が常に不足しているユーザは、知らないうちにマルウェアをインストールしている可能性が高い。

また、Androidマーケットの運営者はマーケットを適切に管理しているわけではないという実情がある。例えば、開発者が開発したアプリケーションをGooglePlayに新しく公開したい場合、Google社はBouncerと呼ばれるマルウェア分析サービスとコンテンツ ポリシー[17]に基づいた審査を行う。しかし、Google社は開示された情報から、ユーザー自身が判断を下すことが重要であると考えているため、審査は最低限しか行われない。そのため、マルウェアやPUPなどの侵入を許してしまっている。また公開されているすべてのアプリケーションを対象としたチェックを行っていないため、動作不良のアプリケーションや迷惑な広告を表示するアプリケーションが放置されている。アプリケーション審査とマーケット全体のチェックの状況は、他のAndroidマーケットについても同様である。以上から、Androidセキュリティの課題として次のことが挙げられる。

- ユーザは十分な情報に基づいた意思決定によって、アプリケーションをインストールしているわけではない。
- Androidマーケット運営者によって、マーケットは適切に管理されているわけではない。

3.2　紹介した既存研究の課題

2.2節では、既存研究をAndroidマーケット及びAndroidアーキテクチャのどのコンポーネントに焦点を当てた研究か、分類して紹介した。ここで別の見方として、Androidにおける静的解析

と動的解析という視点から既存研究を分類する。Android における静的解析とは、Android アプリケーションを逆コンパイルして中のソースコードやマニフェストファイルを解析する手法である。この解析に手法を取り入れている既存研究は [8]、[9]、[10]、[12] が当てはまる。Android アプリケーションは APK ファイルという形式で配布されている。そして、APK ファイル内にある dex ファイルを Jar ファイルに逆コンパイルするツールが無償で配布されている。そのため、APK ファイルからソースコードやマニフェストファイルを取り出すことは容易である。故に自動化しやすく、大量のアプリケーションから解析対象となる情報を入手、解析を行うことができる。静的解析は自動化しやすいという利点がある一方で、実際に動かして得た情報に基づいて解析していないため、解析の対象となったアプリケーションの実態がわからないという問題がある。

次に Android における動的解析とは、実際に指でアプリケーションが表示する画面を操作したときに生じるシステムログやトラフィックを基にアプリケーションの解析を行う手法である。この手法に分類される既存研究は [11]、[14]、[15]、[16] が当てはまる。静的解析とは違い、Android アプリケーションを実際に動かして、その際に生じる情報を解析するため、アプリケーションの実態を掴むことができる。しかし、Android アプリケーションはタッチパネル画面を指で操作するように開発されているため、アプリケーション操作の自動化は難しい。それ故、動的解析に関する手法を用いた提案の実証実験は、サンプルの数が非常に少ないものが多く、実験結果が偶然もしくは意図的に導きされたものではないということを証明できないという問題がある。以上から、静的解析と動的解析から見た既存研究の課題は次の通りである。

- 静的解析の手法を提案する既存研究は、大量のアプリケーションを用いて、提案手法の有効性を実証することができる。しかし、実際に動かしていないため、アプリケーションの実態はわからない。
- 動的解析の手法を提案する既存研究は、アプリケーションの実態を明らかにできる。しかし、アプリケーション操作の自動化が困難なことから、実験に用いるサンプル数が少なくなる。そのため、実験結果が偶然もしくは意図的に導き出されたものではないことを証明できない。

3.3 FireMarker の要件定義

3.3.1 FireMarker の利用者

3.1 節の Android セキュリティの課題と 3.2 節の既存研究の課題から FireMarker の潜在的利用者は 3 種類に分けられると考える。それを次の様に定義する。

Binarian

Binarian はアプリケーション操作から得られる情報のみを要求する利用者である。例えば、Android を専門とする研究者や開発者が Binarian に当てはまる。アプリケーション操作から得られる情報は、例えば、システムログ、外部サーバとの通信で生じるトラフィック、画面遷移の順番やイベントの条件などである。彼らはこれらの情報を新たな動的な解析手法の提案のためのデータセットとして利用すると推測される。また、仮に大量のアプリケーションから動的な情報を得るシステムがあれば、Binarian 自身が考えた新たな動的な解析手法を組み込んだカスタム OS やアプリケーションを用いて、情報を収集する。

Analyst

Analyst は現存する Android マーケットの集約された情報を要求する利用者である。例えば、サイバーポリスやマーケッターが Analyst に当てはまる。集約された情報とは、例えば、Android マーケットの月毎の増加率やアプリケーションの人気とダウンロード数の関係などである。彼らは PlayDrone[8] が示した GooglePlay の情報だけでなく、サードパーティマーケットの集約された情報を望んでいる。また、彼らは複数の Android マーケットの特徴が数値化すれば、それぞれのマーケットの類似性や相対的な安全性、全体を通して利用が多い広告モジュールの種類などを統計的に明らかにし、マーケットの動向を把握したいと考えている。

End-user

End-user は Android マーケットもしくは Android アプリケーションの安全性に関する情報や感染した場合の対処法の情報、またどういう動きをするアプリケーションか、といったことを要求する利用者である。これは専門家ではない一般ユーザが当てはまる。彼らは Android が持つパーミッション機構を理解しておらず、脅威に晒されている。市販のセキュリティ対策ソフトウェアを導入している人もいるが、できればインストール前にアプリケーションの動作を確認したいと考えている。

3.3.2 FireMarker が満たすべき条件

Binarian は FireMarker を次の操作（図 3.1）を行うと推測する。1 つ目は、FireMarker に実装されている標準の動的解析システムを用いて、アプリケーションから情報を収集する操作である。2 つ目は、FireMarker に彼らが提案する手法を実現するために開発したプログラムを組み込んで、アプリケーションから情報を収集する操作である。この操作により、Binarian は標準設定の Android 端末上で得たアプリケーション情報と提案する手法を用いた得た情報の比較をすることができる。

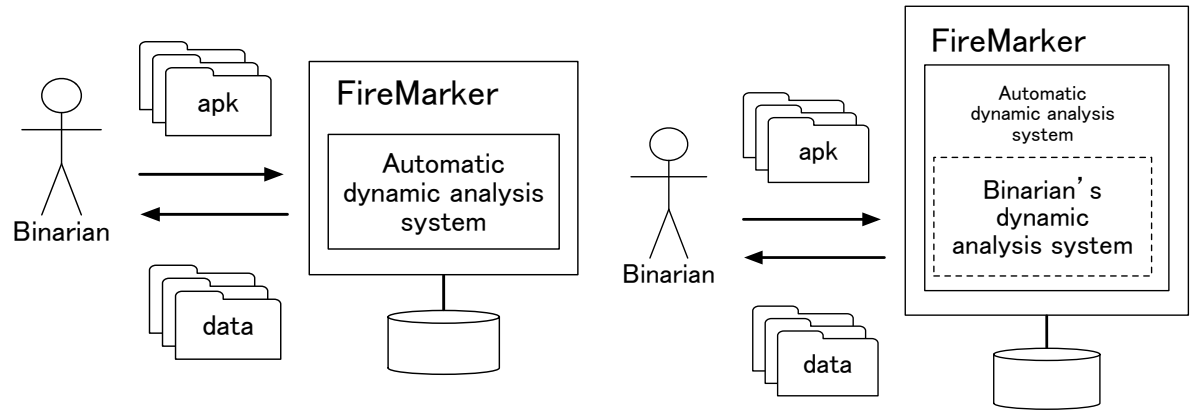

図 3.1: Binarian の FireMarker に対する操作

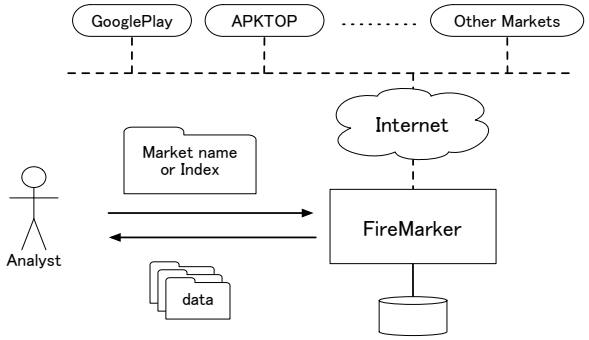

図 3.2: Analyst の FireMarker に対する操作

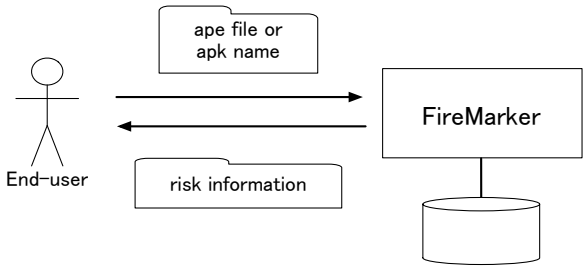

図 3.3: End-user の FireMarker に対する操作

また彼らは実時間以内に大量のアプリケーションから情報を得たいと考えている。Android 端末 1 台を用いて 100 万アプリケーションから起動後 1 分の間に生じる情報を集めるとした場合、単純計算で約 2 年かかるところを現実的な時間に抑えたいと望んでいる。このことから、FireMarker は動的解析の実験時間が非常に長い問題を解決するため、複数のマシン、複数の Android 端末を制御して、大量のアプリケーションから情報を収集する機能が必要である。

次に Analyst は FireMarker に対して 2 通りの操作を行う（図 3.2）と推測する。1 つ目は、FireMarker に Android マーケットの名前もしくは URL を入力し、保存されている各マーケットの傾向や特徴を得る操作である。2 つ目は、FireMarker に Android マーケットへ定期的な観測を指示し、観測された結果を得る操作である。これらの操作によって、Analyst は Android マーケットの現在の特徴や傾向だけでなく、リアルタイムで変わるマーケットの動向に関する情報を得ることができる。この 2 操作を実現するために、FireMarker は Android マーケットを定期的にクロールし、アプリケーションを収集することが必要と考える。加えて、利用者がいない場合でもアプリケーションから情報を集め、まとめる機能がいる。

最後に End-user は FireMarker にアプリケーションファイル、もしくは名前を入力し、アプリケーションのリスク情報、動作の動画などを得ると推測する（図 3.3）。End-user は調べたアプリケーションには利用リスクがあるのか、また利用リスクがどの程度あるのかを知りたがっている。よって、End-user にリスク情報を提示する際には、アプリケーションそのものの情報ではなく、どの Android マーケットに属しており、提示したマーケットにどの程度のリスクがあるかをユーザに伝えた上で利用するマーケットの時点からリスクを意識し、一時的にでもアプリケーションの導入を躊躇させる仕組みがいる。

以上から、FireMarker が利用者の要求に応えるには、次の条件を満たす必要があると考える。

- FireMarker は複数の PC、複数の Android エミュレータ/デバイスを利用して、アプリケーションから情報を収集する。エミュレータ/デバイスはデフォルトでもカスタマイズしたものでも利用できる。収集する情報はアプリケーション操作によって生じるものに限る。

- FireMarker は複数の Android マーケットを定期的にクロールする機能を有する。集めたアプリケーションを自動的に分析し、マーケット毎の特徴を数値化、データベースに蓄積する。利用者はそのデータベースに利用したいキーワードを入力することで、それに合った情報を FireMarker から入手できる。

- FireMarker は Android マーケットの利用リスクを基に、アプリケーションの評価を行う。

以上の条件を基に、設計と実装を行った。4 章において、その詳細について述べる。

第4章　FireMarkerの設計と実装

4.1　全体像と構成

前章で導いた条件から、FireMarker の全体像を図 4.1 に示す。FireMarker は Android マーケット調査システム MarketDrone とマーケットリスク出力システム FireMarking から構成される。MarketDrone は Binarian 及び Analyst の要求に応えるため、複数の Android マーケットをクロールする機能と複数のマシンと複数の Android エミュレータ/デバイスを利用して、Android アプリケーションから動的な情報を収集して、フィルタリングする機能を持つシステムとして設計した。FireMarking は End-user の要求に応えるため、MarketDrone の出力から、Android マーケットの利用リスクを測る機能を持つシステムとして設計した。2 つのシステムを独立させた理由は、Android マーケットの調査によって得られる結果は、様々な視点から分析、評価できる。Binarian の要求から、彼らが MarketDrone が出力した結果を提案する手法で解析できるようにするため、システムを分割した。4.2 節より、MarketDrone と FireMarking の概要を述べる。

4.2　Android マーケット調査システム:MarketDrone

まず初めに、MarketDrone、Android マーケット調査システムについて説明する。このシステムは、4 つのモジュールから成り立ち、それぞれを Crawler、Dispatcher、Filter、Controller と呼ぶ。

4.2.1　Crawler

Crawler は複数の Android マーケットに対応するため、各マーケットの専用クローラからなる集合モジュールとして設計した（図 4.2）。Crawler のメインシステムは Crawler controller と呼び、各マーケットの専用クローラを管理している。また Crawler controller は MarketDrone のモジュールである Controller からの通信の受付口となっている。外部の Controller モジュールの通信によって、各 Android マーケットの自動クロールを行う。

ユーザが Crawler に Android マーケット名もしくは URL を入力すると、Crawler controller がマーケット名及び URL に対応する専用クローラを呼び出し、Android マーケットをクロール、アプリケーションをダウンロードする。そのダウンロードしたアプリケーションは指定、もしくは Crawler が指定する保存先に保存する。各専用クローラは実行すると、すべてのページをクロールするま

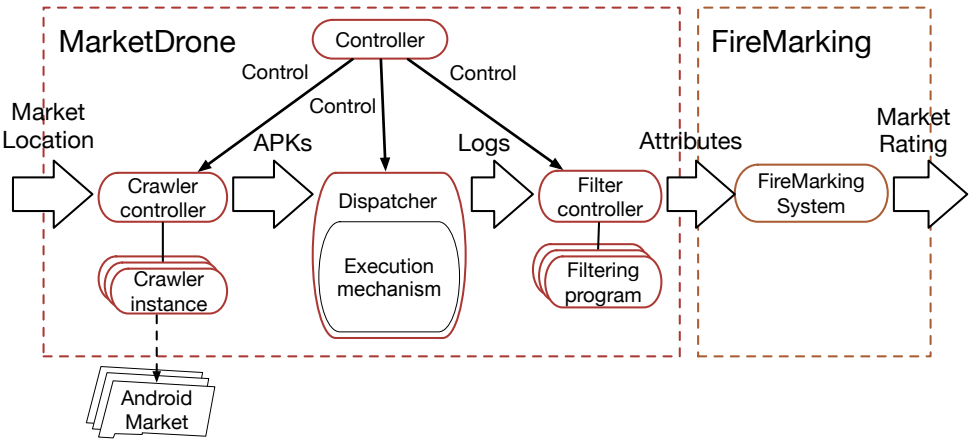

図 4.1: FireMarker の構成

で終了しない。また保存先と差分を取り、新規ファイルであれば、ダウンロードする。特定のジャンルに属するアプリケーションのみダウンロードするといった機能は持たない。

4.2.2 Dispatcher

MarketDrone のメインモジュールである Dispatcher はエミュレータ/デバイスを使い、アプリケーションの最初のスタート画面を表示させ、その際に生じる情報を収集する機能を持つ。ここでは図 4.3 の Execution mechanism の詳細を述べる。

Dispatcher はエミュレータ/デバイスに対して、アプリケーションインストール、スタート画面表示、待機、終了、アンインストール操作を順番に行う。アプリケーションのインストールからスタート画面を表示、待機している間に生じるシステムログとトラフィックを収集の対象としている。

アプリケーション操作は、アプリケーションを起動することのみにした。これは Permlyzer [10] の Android マルウェアは起動時の最初の画面のバックプロセスで危険な権限を要求しているという結果を参考に決定した。

収集する対象をシステムログとトラフィックにした理由は、システムログには、パーミッション要求やアプリケーション内部で利用される命令文のやりとりや強制終了の原因などが出力される。それに加え、TaintDroid [14] などの Linux カーネルに解析ツールを組み込んだ OS を導入したエミュレータ/デバイスを利用する場合、解析ツールが出力する結果はシステムログに表示されるためである。トラフィックは、HTTP プロトコルを収集対象と想定している。これは不審な Web サーバとの通信もしくは広告に関する通信を取得し、秘匿情報を外部に漏らしていないか、調べるためである。

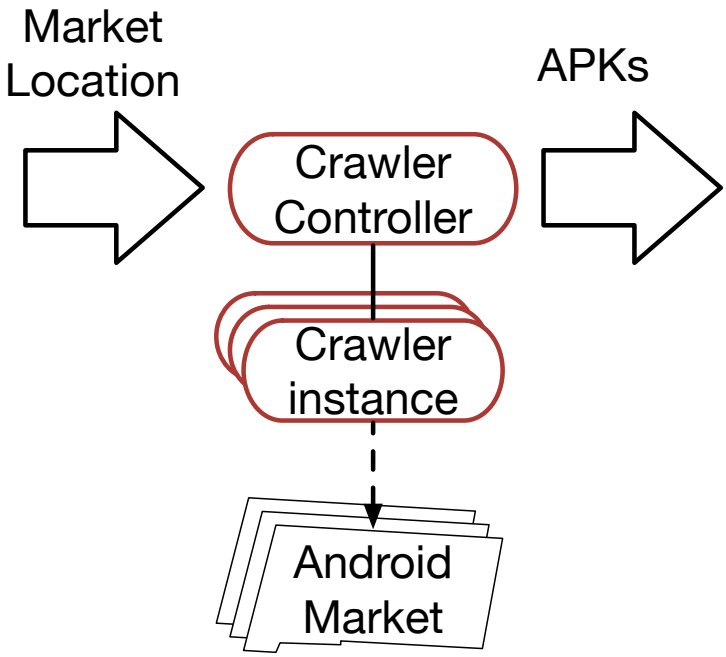

図 4.2: Crawler の設計

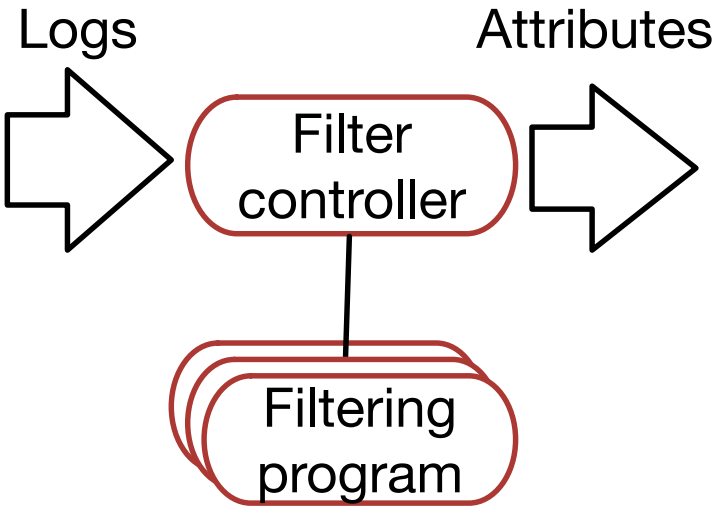

図 4.3: Dispatcher の設計

アプリケーション起動後の待機時間は、利用する Android 端末がエミュレータのみの場合は 30 秒、デバイスのみの場合は 60 秒と設定した。これはアプリケーションインストールに要する時間がエミュレータとデバイスで異なるためである。エミュレータの場合は 1 秒から 10 秒、デバイスの場合は 10 秒から 30 秒かかることが多いことから、それぞれの時間を決定した。

次に Dispatcher の設計について説明する。設計は 2 パターンあり、それぞれを設計パターン A、設計パターン B と呼ぶこととする。設計パターン A は Dispatcher の開始から終了までの時間を短くすることを目的とした設計である。一方、設計パターン B は正常にアプリケーションから情報を集めることを重視し、全体の可用性を高めた設計である。初めに設計パターン A と B の共通点と違いについて説明する。次に設計パターン A、B、各々に基づいた Dispatcher の全体設計と各モジュールについて述べる。

設計パターン A と B の共通点と違い

まず両者の共通点はアプリケーションからシステムログ及びトラフィックを取得する工程である。どちらもエミュレータ/デバイスに対して、アプリケーションインストール、スタート画面を起動、待機、アプリケーションを終了、アンインストールを順に実行し、情報を集める。

システムログの収集には adb コマンドの adb logcat を利用する。トラフィックの収集には外部ツールである http proxy と tcpdump[18] を利用する。Dispatcher の実行時に http proxy を起動し、エミュレータ/デバイスのフローを制御する。データ取得時には tcpdump を制御し、http proxy に送信されるトラフィックを取得する。また、処理の対象となるアプリケーションや取得した情報の保存先はローカルホストではなく、共通のファイルサーバを想定している。これは複数のマシンと連携して処理を行うとき、入力及び出力情報を集約するためである。

一方、両者の異なる点は、Dispatcher と Android エミュレータ/デバイスとの接続と未処理アプリケーションの管理の方法である。設計パターン A（図 4.4）は第 2 章の図 2.3 の adb の接続図に Dispatcher モジュールを加えたものである。Dispatcher は複数の adb client から 1 つの adb server を介して複数の Android デバイスにアプリケーション及び命令を送信し、情報の収集を行う。

この接続の構造の利点は、Dispatcher と adb は互いに独立したプログラムのため、各物理マシンに Android SDK を導入すれば、モジュールをすぐに利用できる。欠点としては、Dispatcher と Android デバイス間で何らかの障害が発生したとき、復旧し難い構造が挙げられる。adb server はマシン 1 台につき、1 プロセスのみ起動するプログラムなため、例えば、Dispatcher が 2 つのデバイス A、B と接続して処理を行い、Dispatcher とデバイス A との接続に何らかの障害が発生し、再接続を試みるとき、adb server を再起動すると、Dispatcher とデバイス B との接続も切断されてしまう。つまり、adb server が止まると、すべての Dispatcher とデバイス間の処理に大きな影響を及ぼすことになる。

設計パターン B（図 4.5）は設計パターン A を改良し、Dispatcher とエミュレータ/デバイス間の接続障害に強くした設計である。この設計では、Dispatcher は 2 つの方法でデバイスと接続している。1 つは図 4.4 と同じく、adb と USB を利用して接続する。もう 1 つは、Dispatcher とデバイスの間に仮想マシンを挟み、ホストマシンにある Dispatcher の代わりに仮想マシン内にある Dispathcer が adb と TCP 通信を使って、デバイスと接続する方法である（以下、ホストマシンにある Dispatcher を parent、仮想マシンにある Dispatcher を child と呼ぶこととする）。仮想マシン、デバイス、child、adb server を 1:1:1:1 にすることで、child とデバイスの接続に障害が起こったとしても、復旧が容易になる。反面、可用性を重視したため、設計パターン A よりも処理に費やす時間が長くなる恐れがある。また、起動させる仮想マシンの数だけホストマシンの資源を消費するので、全体の処理が重くなる可能性もある。

続いて未処理アプリケーションの管理について述べる。設計パターン A では入力された未処理アプリケーションをエミュレータ/デバイスの数だけ均等に分割し、エミュレータ/デバイスに割り振っている。アプリケーションを均等に割り振っているので、Dispatcher による処理の終了時間を推定しやすいことと、エミュレータ/デバイスの数が多ければ多いほど Dispatcher の総処理時間は短縮されると考えている。しかし、処理速度を重視しているため、予期せぬエラーに弱い。例えば、Dispatcher とエミュレータ/デバイス間の接続が切れ、処理が停止した場合、そのエミュレータ/デバイスが担当するアプリケーションの情報収集を放棄しなければならない。その放棄したアプリケーションは 2 回目以降の実行時に再度処理を行う。

設計パターン B は parent が未処理アプリケーションを管理しており、child がそれらの中から 1 アプリケーションを受け取りに行く方法を採用している。これによって、複数 child のうちの 1 つが処理を停止したとしても、他の child には影響を及ぼさず、Dispatcher の処理を継続できる。一方、処理速度は重視していないため、実行終了時間は予想よりも長くなる可能性がある。

設計パターン A の全体設計

設計パターン A に基づいて設計された Dispatcher はエミュレータ/デバイスとの接続処理、未処理アプリケーションの管理、エミュレータ/デバイスを用いて行う情報収集のための処理などを全て行う（図 4.6）。処理の流れは次の通りである。

（1）ユーザは Dispatcher に Android アプリケーションがあるディレクトリと Android エミュレータ/デバイスのシリアル番号を入力し、Dispatcher を実行する。

（2）入力後、Dispatcher はエミュレータ/デバイスのシリアル番号をもとに、adb を介して各デバイスに接続する。接続したことを adb device を使い、確認する。

（3）接続できたエミュレータ/デバイス数に応じて、アプリケーションを振り分ける。

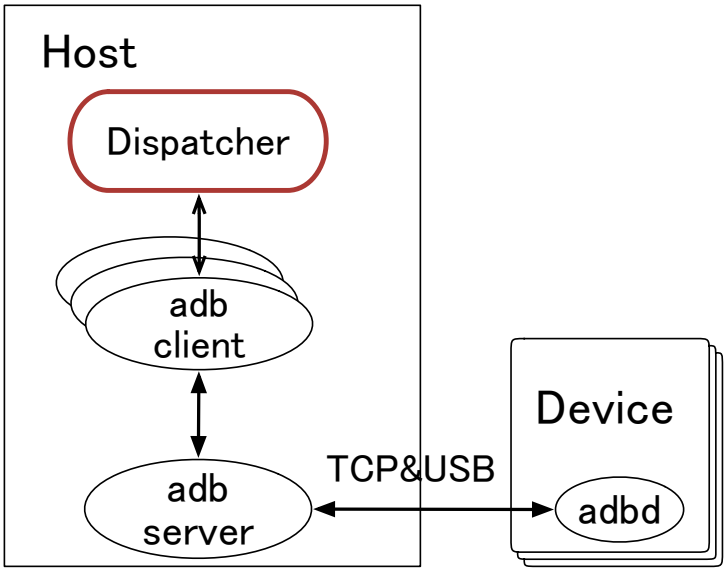

図 4.4: 設計パターン A

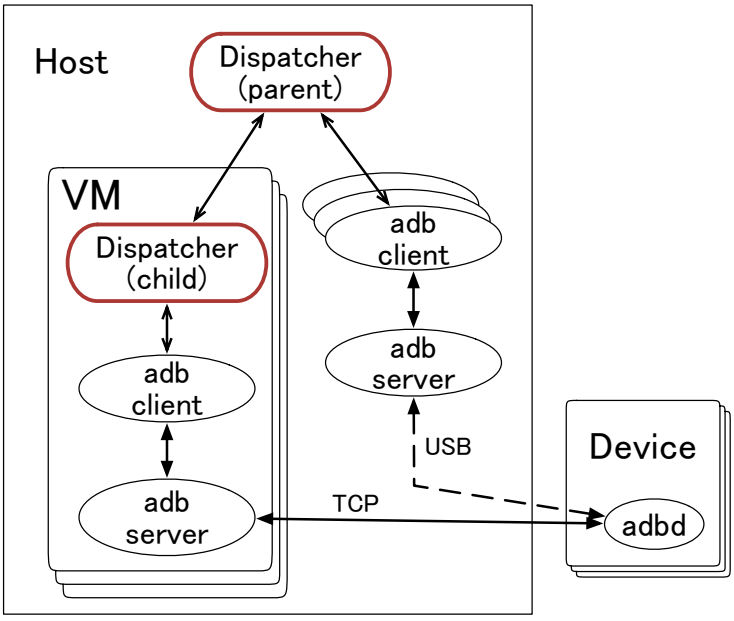

図 4.5: 設計パターン B

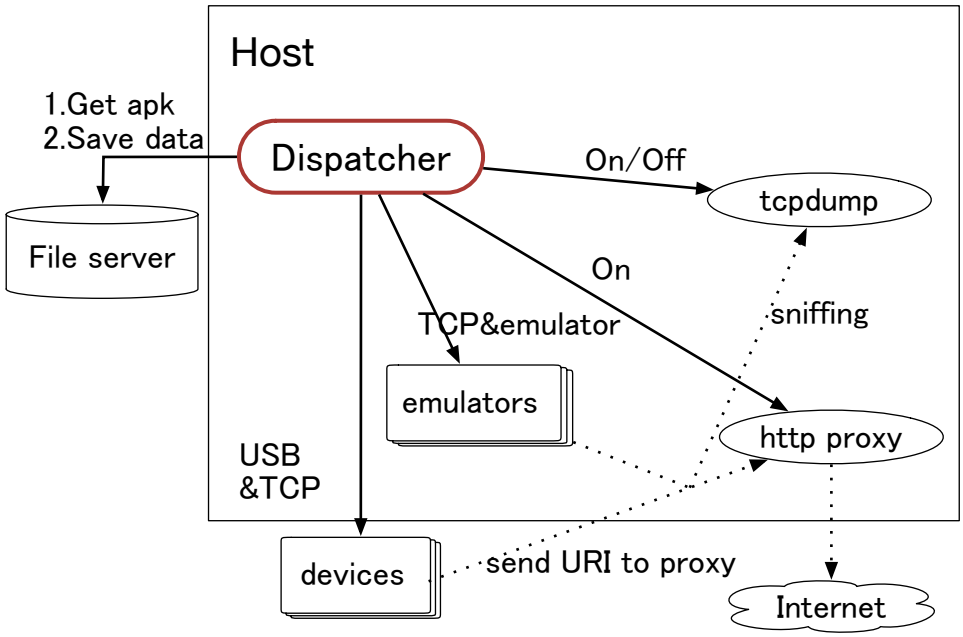

図 4.6: 設計パターン A による Dispatcher の構成

（4）アプリケーションのインストール、実行、終了、アンインストールを端末毎に行う。

（5）インストールから実行までの操作を adb logcat、tcpdump を利用して、デバイスの場合は 60 秒間記録し、エミュレータの場合は 30 秒間記録する。エミュレータ/デバイスで取得時間が異なるのは、アプリケーションのインストールにかかる時間が異なるためである。

（6）以後、（4）と（5）を繰り返す。

（7）Dispatcher に入力したすべての Android アプリケーションに（4）、（5）の操作を行ったとき、Dispatcher を終了する。

また、設計パターン A に基づいた Dispatcher を複数のマシンで利用する構成は図 4.7 としている。Dispather の上位モジュールとして、Dispatch manager がある。これは複数のマシンにある Dispatcher の実行、終了、入力されたアプリケーションの割り振り等を制御する機能を持つ。

設計パターン B の全体設計

設計パターン B に基づいて設計された Dispatcher は parent、child、そして relation に分かれる（図 4.8）。モジュールの起動は parent から実行され、複数マシンと連携する場合は relation、最後

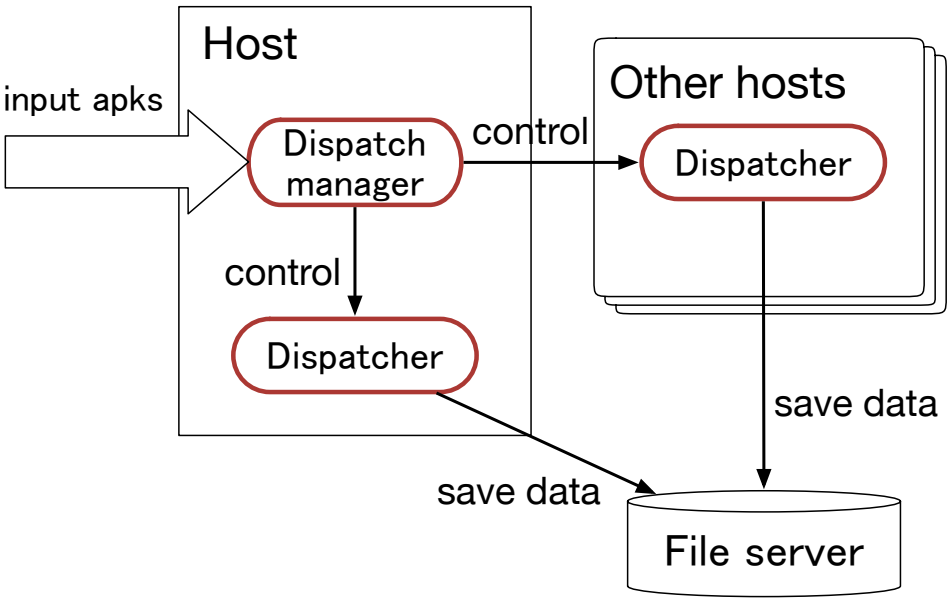

図 4.7: 複数のマシンを利用する場合の構成

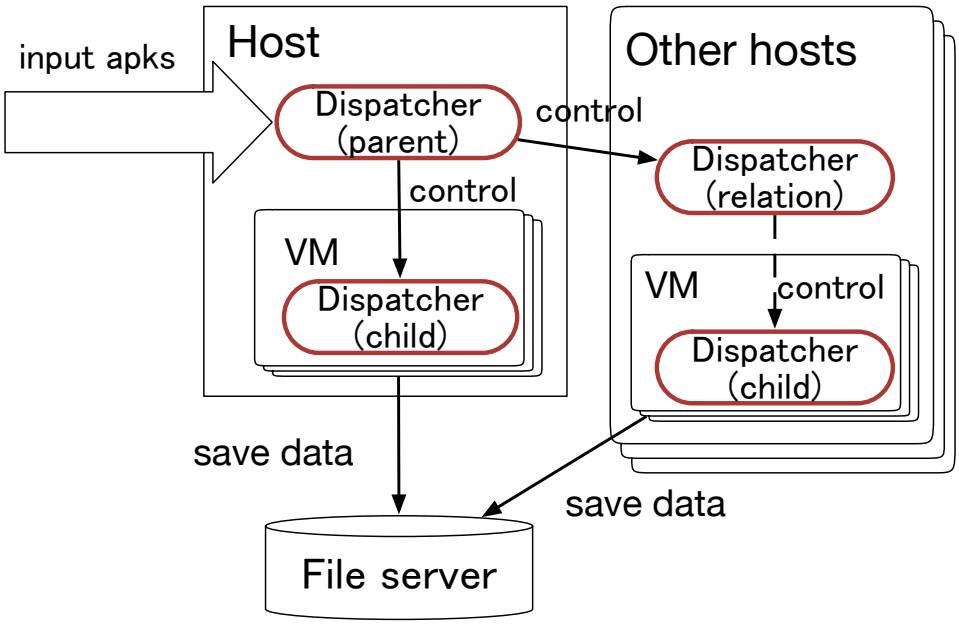

図 4.8: 設計パターン B の全体像

に仮想マシン上の child が起動する。次に Dispatcher を構成する parent、child、relation を順に説明する。

parent（図 4.9）は、未処理アプリケーションを管理する APK manager、仮想マシンを制御する VM manager、接続されているエミュレータ/デバイスを管理する Devices manager という 3 つのモジュールとそれらを管理する Main controller モジュールによって構成されている。APK manager は起動時にファイルサーバ上の収集対象のアプリケーション名を保持し、child からの要求に対して、アプリケーション名を渡す機能がある。VM manager は仮想マシンの起動や child の実行、http proxy の実行等、仮想マシンに関する処理の全てを行う。VM manager が起動する仮想マシンは Android が既にインストールされており、ファイルサーバ内にある収集対象を保存したフォルダと取得した情報を保存するフォルダを仮想マシン上の指定したディレクトリにマウントしたマシンである。また、立ち上げる仮想マシンの数はエミュレータ/デバイスの数と同じである。

Devices manager はエミュレータ基盤を監視、もしくは USB ケーブルを介してデバイスを監視している。またエミュレータ/デバイスの接続数を調べ、VM manager にその数を送る。child とエミュレータ/デバイス間に接続障害が起きた場合、child の復旧要求に従い、問題となるエミュレータ/デバイスの再起動や復元を行う。複数台のマシンを用いる場合は、Main controller が relation に対して実行命令を送る。

child（図 4.10）は Dispatcher のメインモジュールである。ここではアプリケーションからシステムログ及びトラフィックを取得する機能を持つ。child は VM manager の起動命令により起動する。そして、APK manger から対象のアプリケーション名を要求する。受け取った名前をファイルサーバから探し、エミュレータ/デバイスにインストール、情報の取得を行う。エミュレータ/デバイスとは adb を介して、TCP 通信によって接続している。取得した情報は同じくファイルサーバ内の指定したフォルダに保存する。child は VM manager が起動した仮想マシンの数だけ存在する。

relation（図 4.11）は parent から APK manager を除いたものであり、relation 内部の各モジュールは parent と同様のことを行う。relation は parent の起動命令によって、実行される。

4.2.3 Filter

Filter は Dispatcher によって得られた情報を任意の基準のもと選別する複数の専用ツールからなる集合モジュールとして設計した（図 4.12）。Filter のメインモジュールは Filter controller と呼び、専用ツールの管理を行う。また Filter controller は Controller からの通信の受付口となっている。ユーザが Filter に特定のプロトコル名やキーワードとシステムログファイル、pcap ファイルを入力すれば、それを抽出する。また結果をグラフ、表にまとめる機能を持つ。出力した結果は File server の指定したフォルダに保存する。

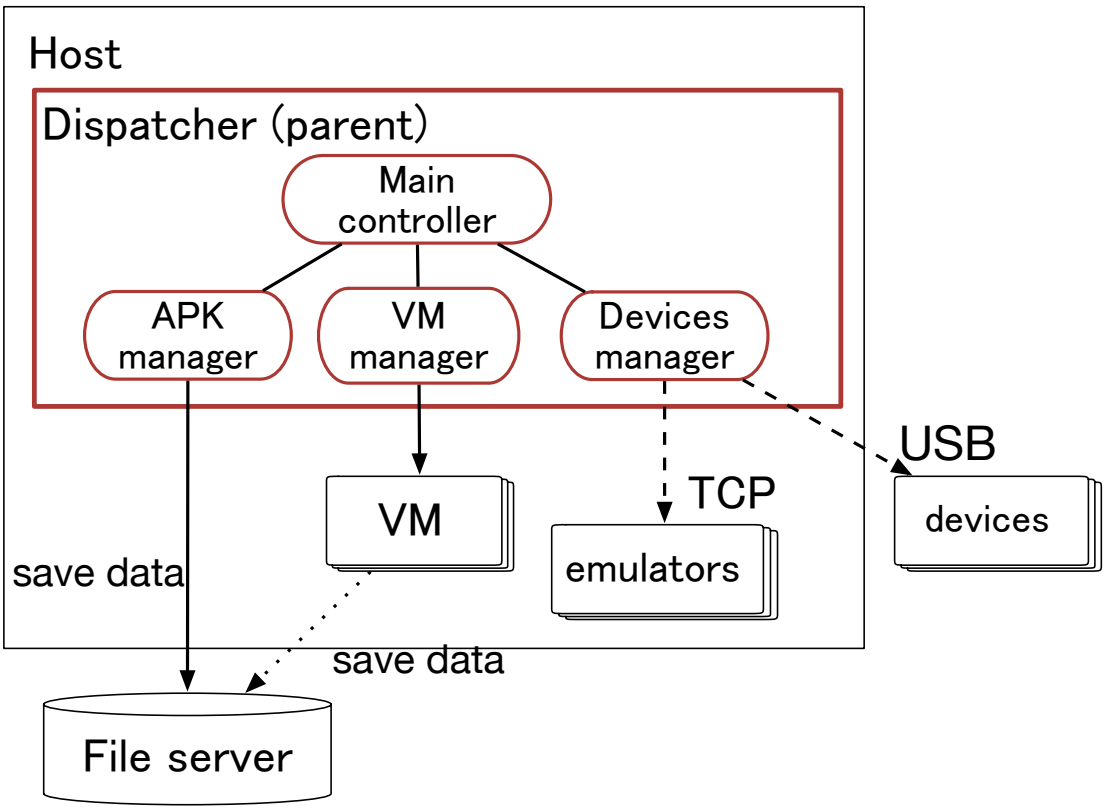

図 4.9: parent の構成

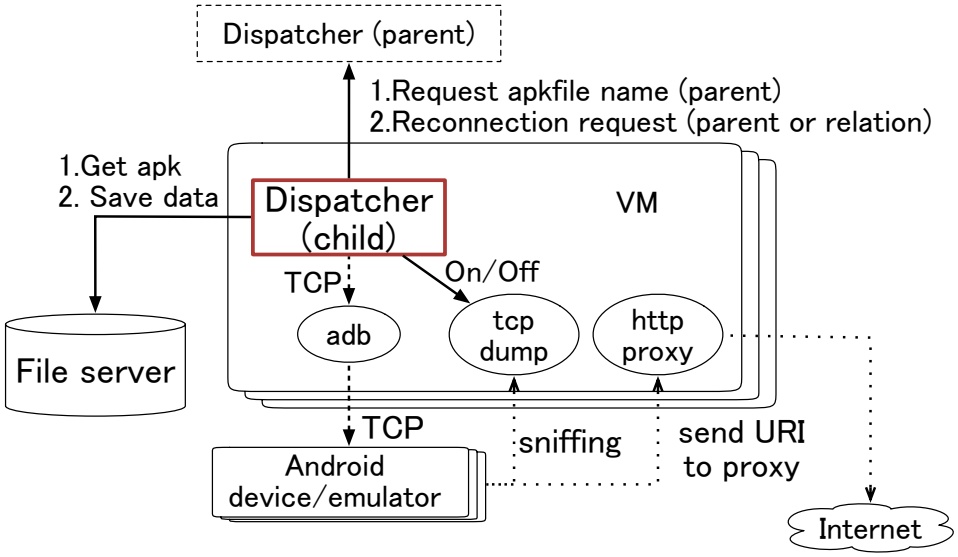

図 4.10: child の構成

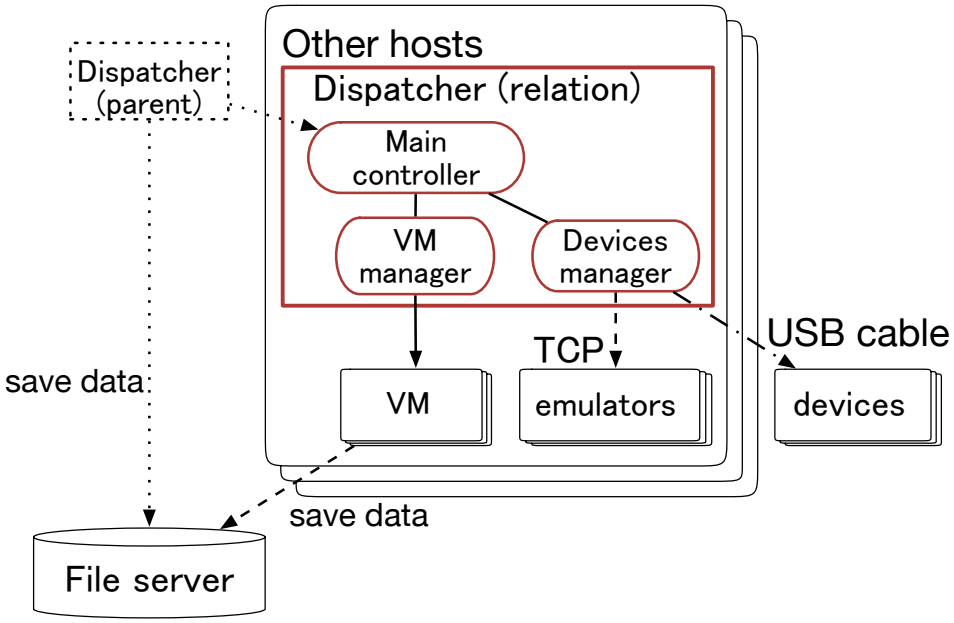

図 4.11: relation の構成

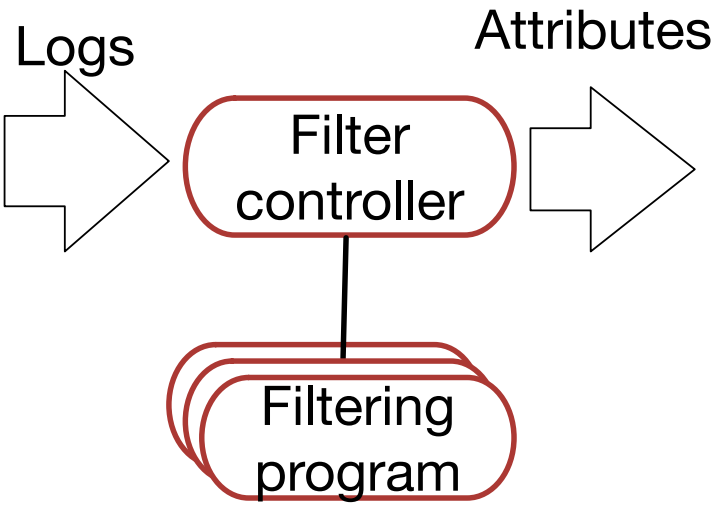

図 4.12: Filter の設計

4.2.4 Controller

MarketDrone の各モジュールは互いに独立したプログラムであり、ユーザは何れかを選択して利用できる。Controller は全てのモジュールを統合し、複数の Android マーケットを定期的に調査する機能を持つ。これはモジュールの実行と終了、データの保存フォルダ先を監視し、互いのプログラムが干渉しないように制御する。

4.3 マーケットリスク出力システム:FireMarking

FireMarking は MarketDrone が出力したデータを基に、各アプリケーションの利用リスクを測る。FireMarking の設計と実装について、第 6 章で述べる。

4.4 FireMarker の実装

ここでは MarketDrone 及び FireMarking の実装したものについて述べる。MarketDrone は 4 つのモジュール、Crawler、Dispatcher、Filter、Controller のうち、Crawler、Dispatcher、Filter のプロトタイプを実装した。しかし、各モジュールのメインシステムの実装を優先したため、Crawler、Dispatcher、Filter を統合し、自動化するモージュル Controller や Crawler、Filter 内にある Controller といった、MarketDrone の自動化を支援するモジュールは実装していない。

4.4.1 Crawler

Android マーケットである、APKTOP[19] をクロールする専用ツール apktop crawler を実装した。このマーケットは、配布 Web サイトの構造が簡易で、ダウンロードの際に特別な認証をする必要がなく、かつ 1 万以上のアプリケーションを公開していることから、実装した。

4.4.2 Dispatcher

考えた設計パターン A、B のどちらも実装した。設計パターン B で使う仮想マシンは Linux Containers（以下、LXC と呼ぶ）を用いることとした。LXC とは 1 つのコントロールホスト上で、複数の隔離された Linux システムを実行する、OS レベル仮想化ソフトウェアであり、プロセスやネットワーク、ユーザー空間などを分離して、仮想的な環境（以下、LXC インスタンス）を提供する技術である。

LXC の利用は、カーネルの機能を使って分離された環境であるため、LXC インスタンスでは CPU やディスク I/O などのパフォーマンスが影響しないこと、また、隔離された環境の一部を外す

設定によって、マシンとマシン内にある LXC インスタンスとの間に共有フォルダに作れることから採用を決めた。設計パターン B の parent 内の Devices manager が child の要求によって行うマシン/デバイス間の再接続の処理時間は約 2 分と定めた。そのほか、経験則からプログラム内で adb の shell コマンドを実行したり、エミュレータ/デバイスに接続要求した後は 1 秒間停止するようにした。

4.4.3 Filter

Filter は専用解析ツールを 4 つ実装した。それはシステムログからユーザに管理者権限を要求するアプリケーションを特定する rootchecker、システムログから強制終了したアプリケーションを特定する forcechecker、トラフィックから HTTP 通信だけを抽出し、FQDN を数と種類をグラフに出力する FQDNchecker、同じく HTTP 通信を抽出、Adblock[20] のブラックリストと URI を比較し、迷惑な広告や疑わしい挙動をする広告を選出する adchecker である。それぞれのツールは次の理由から、今回実装した。

管理者権限の要求

まず、Android デバイスの Root 化について説明する。Root 化とは、ユーザが Android デバイスのサブシステム内での特権（管理者権限）を取得する作業を指す。この作業によって、ユーザはキャリアやメーカーが Android デバイスに施した制限を解除し、システムアプリケーションや設定を改変もしくは置換することができる。また、管理者権限が必要なアプリケーションを動作させ、通常の Android デバイスでは利用できない操作を行うことができる。これは iOS のジェイルブレイクと同義である。

米国アップル社はジェイルブレイクへの見解として、2009 年 2 月に「ジェイルブレイクが、著作権で保護されているファームウェアを無効にする行為は、アップルの著作権やデジタルミレニアム著作権法に違反する」というコメントをアメリカ著作権局に提出しており、否定的な姿勢を示している [21]。その一方で、Android を提供する米国 Google 社は公式開発者ブログで「Root 化はユーザの当然の権利」と肯定的な姿勢を示している [22]。加えて、米国著作権局は 2010 年に Android デバイスを Root 化し、サードパーティーアプリケーションを動かしたり、携帯電話をアンロックしてキャリアフリーにしたりすることは適法であることを公式に表明した [23]。

しかし私は、ユーザが知識もなしに Root 化すること、管理者権限を要求するアプリケーションを利用することは危険な行為であると考えている。管理者権限の利用を承認されたアプリケーションは、電話帳や Android デバイスに登録されているクレジットカードの情報といった秘匿情報へのアクセスができるようになる。また、ユーザの許可なく、Android 端末を操作することができる。管理者権限を要求するアプリケーションが Android マルウェアであった場合、ユーザの秘匿情報

を外部に流出させられてしまうだけでなく、リモートコントロールされたり、ボットとして DDoS 攻撃に加担させられることにもなりかねない。以上から、管理者権限を要求するアプリケーションは危険であるものと考え、実装した。

強制終了

アプリケーションの強制終了は、Android デバイスのハードウェア性能が基準を満たしていないか、アプリケーション側の致命的なバグである場合が多い。直接的な脅威とはならないものの、Android マーケットの管理の程度を見るために有効な指標であると考え、それを特定するアプリケーションと総数を数え上げる解析ツールを実装した。

FQDN

アプリケーションの HTTP 通信のうち、FQDN に注目して分類することで、その通信が何のために行われているか、どのような内容の通信なのか、などを推測しやすくするために実装した。

広告

Android アプリケーション、特に無料アプリケーションはタッチパネル画面に広告を掲載するものが多い。しかし、実際に何の広告が使われているか、知られていない。広告に該当する HTTP 通信を調べることで、利用されている広告の種類が明らかになり、また、アドウェアといったマルウェアの発見に役立つと考え実装した。

第5章　FireMarkerの実験

本章では前章で説明し、実装したFireMarkerのMarketDroneの実験を行う。この実験によって、MarketDroneの性能と有用性を測る。ここで各実験に用いるマシンの性能は表5.1の通りである。Host Aはすべての実験のメインマシンとして使う。Host BはDispatcherの設計パターンBの実験の際に使うサブマシンである。全体を通して利用したツールは、adb、tcpdump、Privoxy（http proxy）[24]、virtualbox、AndroVM（OSver4.1.1, 現:Genymotion[25]）とNexus7である。マシンとNexus7の物理的な接続には、マシン本体の電力が足りなくなる恐れから、電力供給のあるUSBハブを使用した。

5.1　MarketDroneを用いた実験

この節では、MarketDroneの各モジュールごとの実験について説明する。まず初めにCrawlerの専用クローラ、apktop crawlerの性能実験を行う。次に、Crawlerから得られたAPKTOPのアプリケーションを使い、2通りの設計パターンで実装したDispatcherの性能実験を行う。最後にAPKTOP及びOperaMobileStoreからアプリケーションをダウンロードする専用クローラopera crawlerを用いて集めた10万アプリケーションから、それぞれ1万アプリケーションから情報を集め、Filterによって Androidマーケットの一傾向を明らかにする実験を行う。

表5.1: 実験マシンの各スペック

	Host A	Host B
OS	Ubuntu 14.04 LTS	Ubuntu 14.04 LTS
CPU	Intel Core i7	AMD Processor model unknown
memory	32GB	8GB
GPU	Geforce GTX680	なし

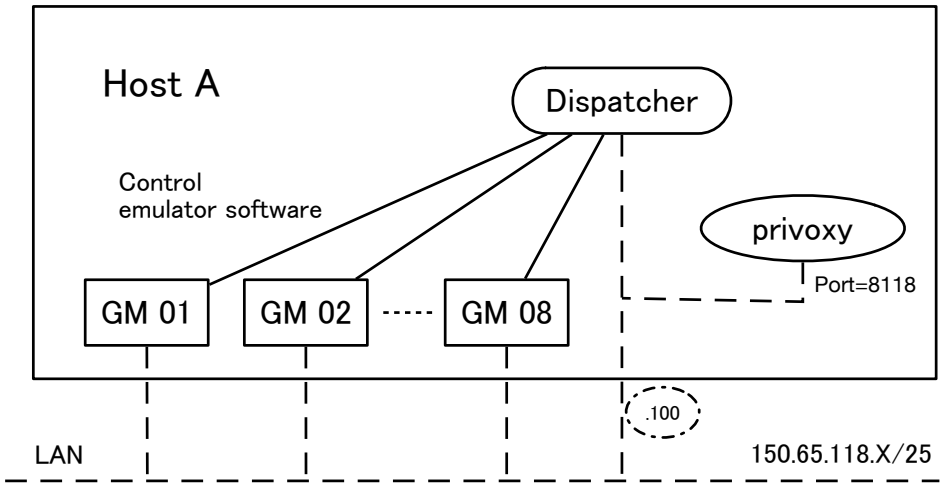

図 5.1: 設計パターン A の実験環境の構成

5.1.1 Crawler の性能実験

この実験は第 4 章で実装した apktop crawler を実行後、終了までの時間を計測する。また、ダウンロードしたアプリケーションの総数、空ファイル数を調べる。これによって、apktop crawler の性能を測る。

実験結果として、apktop crawler は Android マーケット APKTOP から 26,691 アプリケーションをダウンロードした。そのうち、有効ファイルは 25,157、空ファイルは 1,534 あり、apktop crawler は終了までに約 4 日間を要した。このことから、apktop crawler は 1 日に約 7,000 アプリケーションをダウンロードしたことになる。その他、APKTOP という Android マーケットについて、apktop crawler からの標準出力を見た限り、リンク切れしているページが多いことがわかった。

5.1.2 Dispatcher を用いた実験

次に設計パターン A、B それぞれで実装された Dispatcher の性能実験を APKTOP の 25,157 アプリケーション用いて行う。実験方法はそれぞれの Dispatcher に 25,157 アプリケーションを入力し、収集開始から終了までの時間と回収失敗率を測る。設計パターン A の実験環境は図 5.1 である。実験に用いる Android 端末は Android エミュレータ AndroVM[25]8 台（GM01 から 08）を利用した。

設計パターン B の実験環境は図 5.2、図 5.3、図 5.4 である。実験に用いる Android 端末は 7 台の Nexus7（ブートローダーアンロック済み、管理者権限取得済み）を利用した。設計パターン A、B で利用するエミュレータ/デバイスが異なるのは、第 4 章で設計パターン B の実装において採用した LXC と virtualbox との相性に問題があるためである。Host A には 4 台、Host B には 3 台の

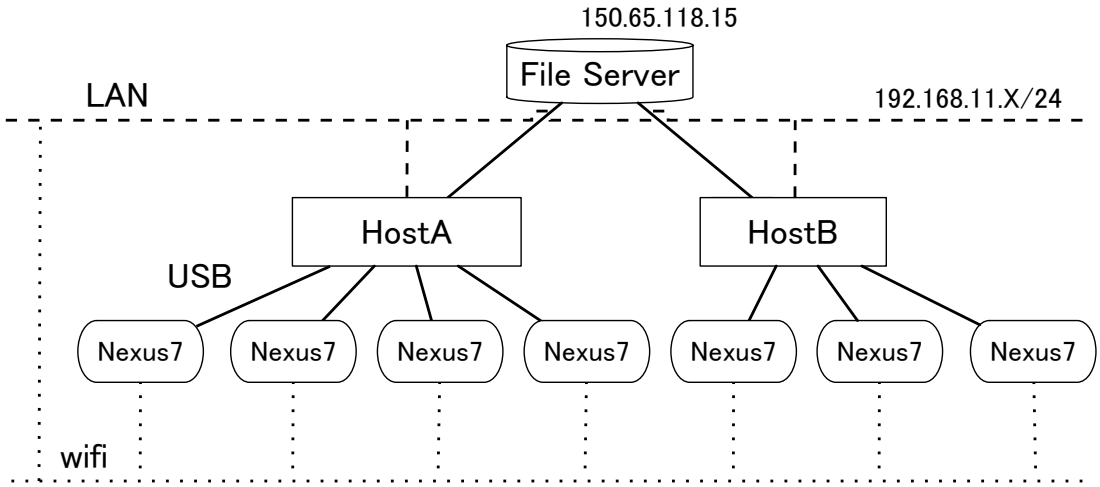

図 5.2: 設計パターン B の実験環境の構成

Nexus7 がそれぞれ USB ケーブルで接続している。また Host A、B 上では LXC インスタンスが 7 台起動している。各 LXC インスタンスには予め Android SDK やそれを利用するのに必要なソフトウェア、http proxy として Privoxy を導入し、静的な IP アドレスを振った。各 Nexus7 には静的な IP アドレスを振り、LXC インスタンス上の Privoxy を経由して Web サーバと繋がる設定を施した。データ収集の対象となるアプリケーションファイルや出力結果は設計通り、ファイルサーバへ保存するようにした。

設計パターン B の処理時間測定

初めに設計パターン B に約 2 万 5 千アプリケーションを入力、実行から終了までの時間を予測するための実験を行った。実験パターンは事前に 1 台のマシンに接続できる限界台数を測り、決定した。この実験は表 5.2 の 7 パターンをそれぞれ 1 回ずつ試行した。1 アプリケーションの理想処理時間は設計から 90 秒/apk と決め、それぞれのパターンの終了時間（表 5.3）を予想した。表 5.4 及び図 5.5、図 5.6 は計測の結果及びグラフにしたものである。

情報収集で利用する Nexus7 の数が増えるほど、全体の処理速度は向上していることがわかる。グラフは一次関数で、情報収集時に利用する端末が多いほど切片が小さくなることがわかった。しかし、表 5.5 のように 1 アプリケーションあたりに費やす時間は Nexus7 が増える毎に増加することがわかった。また理想終了時間（表 5.3）と比べると、最小 300、最大 1,200 秒の差があった。終了時間が遅れた原因を調べたところ、LXC 上の child とデバイスとの TCP 通信の切断が多発しており、その復旧処理で多く時間を要したことがわかった。

図 5.3: 実験環境の写真その 1

図 5.4: 実験環境の写真その 2

表 5.2: 実験パターン

	アプリケーション数		
マシン数-Nexus7 の数	100	500	1,000
1-1	○		
1-4	○	○	○
2-7	○	○	○

表 5.3: 理想終了時間（sec）

	アプリケーション数		
マシン数-Nexus7 の数	100	500	1,000
1-1	9,000		
1-4	2,250	11,250	22,500
2-7	1,286	6,429	12,857

また 1 台のマシンに Nexus7 を 4 台つなげて処理を行ったものと、2 台のマシンに Nexus7 を計 7 台となるように接続し処理をしたものを比べると、1 アプリケーションに費やす時間は 1 台のマシンに Nexus7 を 4 台つなげて処理した場合の方が短いことがわかった。接続エラーの数と利用するデバイスの数が少ないほど 1 アプリケーションに費やす時間は短くなることから、デバイスの数を増やしても、接続障害などのエラーを吸収しきれないことがわかった。これは現在の復旧処理が正常に実行されず、また実行されたとしても正常に接続できないことに原因があることがわかった。その一方で、失敗率（表 5.6）を見ると、アプリケーション数に失敗率は比例するが、デバイス数に比例しないことがわかった。このことから、情報収集に失敗するアプリケーションは特定できる可能性があるとわかった。以上の計測の結果から、設計パターン B に APKTOP、25,157 アプリケーションを入力、実行した場合、約 5 日かかると推測した。

表 5.4: 処理時間測定の結果（sec）

	アプリケーション数		
マシン数-Nexus7 の数	100	500	1,000
1-1	9,738		
1-4	2,527	11,939	23,344
2-7	1,517	6,989	17,655

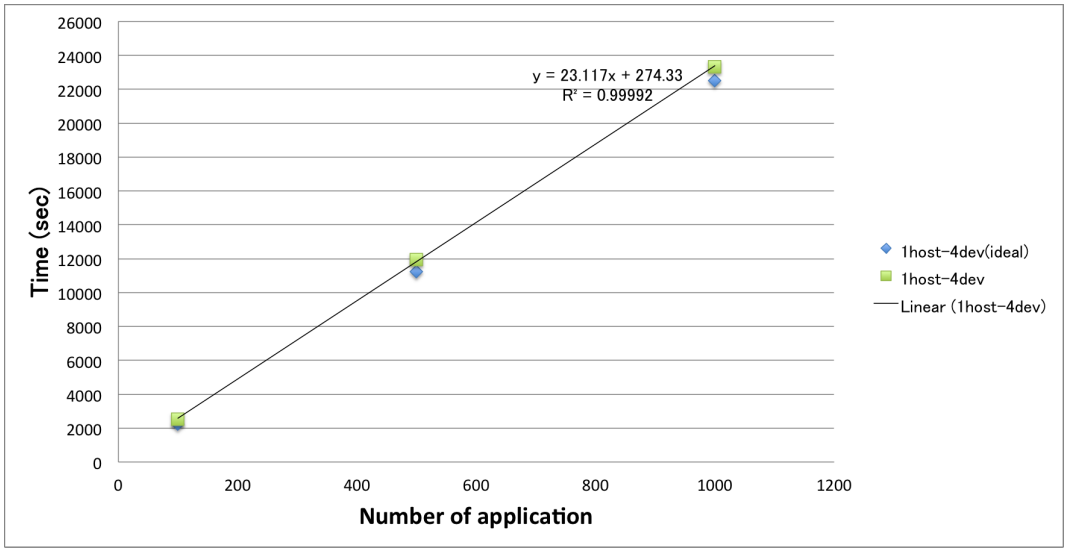

図 5.5: 1 台のマシンと 4 台の Nexus7 の実行時間の推移

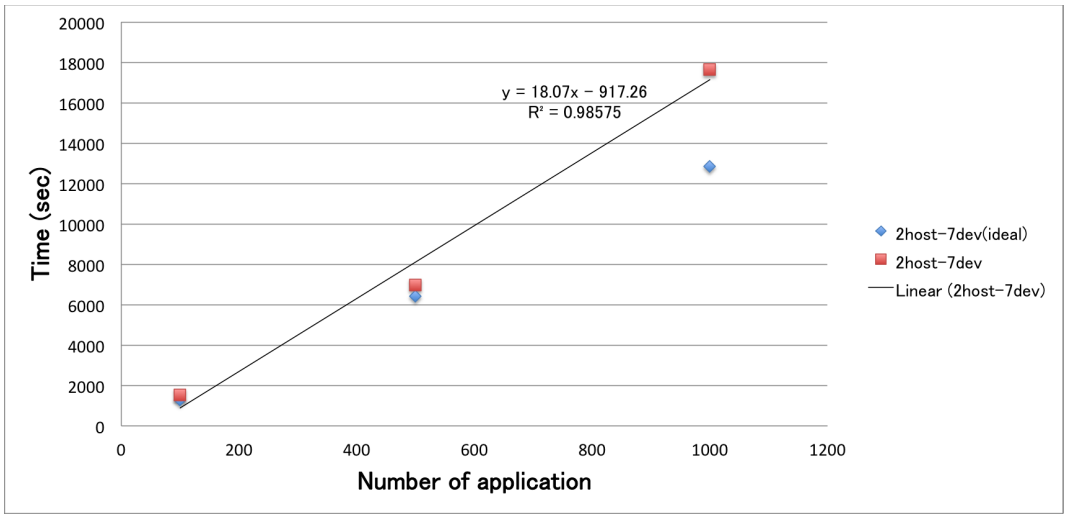

図 5.6: 2 台のマシンと 7 台の Nexus7 の実行時間の推移

表 5.5: 1 アプリケーション当たりに費やす時間（sec/apk）

	アプリケーション数			
マシン台数-Nexus7 の数	100	500	1,000	平均
1-1	97.38			
1-4	101.08	95.51	93.38	96.66
2-7	106.19	97.85	123.59	109.21

表 5.6: 情報収集の失敗率（%）

	アプリケーション数		
マシン台数-Nexus7 の数	100	500	1,000
1-1	2.0		
1-4	2.0	5.4	5.9
2-7	2.0	5.0	5.9

設計パターン A、B の処理時間計測

次に設計パターン A、B それぞれ実装した Dispatcher に APKTOP、25,157 アプリケーションを入力して性能の比較を行った。その結果が表 5.7 である。設計パターン A は 1 アプリケーションに費やす処理時間はアプリケーションから情報を収集する 30 秒とほぼ同じであった。しかし、処理中に 1 エミュレータとマシンとの通信が切れてしまったため、約 3,000 アプリケーションに影響が出た。結果、失敗率が 1 割を超えた。

設計パターン B は全体の処理時間は予想した 5 日よりもやや短く、4 日と 7 時間かかった。1 アプリケーションに費やす処理時間は 103 秒と表 5.5 の 2 台のマシンに計 7 台をつけた場合よりもやや短くなった。失敗率は 6.35%と表 5.6 のアプリケーション数 1,000 個の場合とあまり変わらなかった。この表から設計パターン A は処理時間に優れているが、障害に弱く、設計パターン B は障害に強いが実行時間は長くなる、とわかった。

表 5.7: 設計パターン A、B の性能比較

Dispatcher	実行時間	sec/apk	失敗率（%）
設計パターン A	96,751	30.7	12
設計パターン B	373,135	103.8	6.35

表 5.8: 各マーケットの収集結果

マーケット名	処理時間（sec）	失敗率（%）
APKTOP	133,067	6.42%
OperaMobileStore	146,159	8.41%

5.1.3 Filter を用いた実験

ここでは Filter が持つ 4 つのツールを用いて、APKTOP の約 2 万 5 千アプリケーション及び既存のクローラを用いて集めた OperaMobileStore[26] の約 10 万アプリケーションから意図的に選んだ 1 万アプリケーションを調査する。それぞれ 1 万アプリケーションとした理由は、Dispatcher の性能実験の結果によって、OperaMobileStore の約 10 万アプリケーションから情報を集めるのに必要な時間は約 25 日とわかったためである。

初めに情報収集の失敗率が小さい設計パターン B を用いて、情報を収集した。次に Filter を使い情報を解析した。まず、各マーケットの実行時間と情報収集の失敗率は表 5.8 の通りである。実行時間から 1 アプリケーションにかかる時間は、APKTOP が約 93 秒、OperaMobileStore が約 102 秒とわかった。失敗は、マシンと Nexus7 との接続に障害が生じていたことや対象アプリケーションにメインアクティビティがないため起動できなかったこと、対象アプリケーション名が長すぎて、adb が読み込めなかったことなどが原因とわかった。

次に表 5.9 から APKTOP は次の傾向があるとわかった。

- 管理者権限を要求するアプリケーションが多い。
- HTTP 通信は広告以外のものが多い。もしくは外部と通信しないアプリケーションが多数ある。

また、OperaMobileStore は次の傾向があるとわかった。

- 広告のための HTTP 通信を行うアプリケーションが多い。
- 強制終了するアプリケーションが少ないことから、比較的管理されてたマーケットである可能性がある。
- 管理者権限を要求するアプリケーションがないことから、OperaMobileStore ではこのようなアプリケーションに否定的である。

次にトラフィックから FQDN を抽出し、上位 20 をまとめた結果を図 5.7、図 5.8 にそれぞれ示す APKTOP と OperaMobileStore の上から 5 位までを比較すると、どちらもほぼ同じ FQDN がランクインしていた。これらを調べた結果は表 5.10 である。

表 5.9: Filter による各マーケットの傾向

	システムログ		トラフィック
マーケット名	強制終了（apks）	管理者権限要求（apks）	adchecker のヒット数
APKTOP(10,000)	419	2,241	863
OperaMobileStore(10,000)	246	0	3,638

表 5.10: 上位 5 位に見られる各 FQDN の内容

FQDN	内容
data.flurry.com	モバイルアプリケーション用ユーザ行動解析サービスを展開する Flurry に関係している。
googleads.g.doubleclick.net	Google が配信する広告サービス Google アドセンスに関係している。
media.admob.com	Google の子会社で、モバイルアプリケーション用広告配信サービスを展開する admob 社に関係している。
ssl.google-analytcs.com	Google アナリティクスで利用する。
android.clients.google.com	Google アカウントに紐付けされている情報を要求するとき、例えば Oauth キーを要求する時に利用する。

最後に adchecker の各フィルタのヒット数を表 5.11 に示す。フィルタの内容は表 5.12 の通りである。Japan 2 に該当する Adblock 公式の日本用フィルタにヒットが集中している。次点で General、Japan 1 が続いている。これは Android アプリケーションが位置情報に応じて、適した広告が配信されるため、日本のものが多くフィルタリングされたからであると考えられる。その一方で、中国やロシア、エストニアなどのフィルタにもヒットした。その他にも両マーケットとも Malware フィルタにヒットした（表 5.13）。

それぞれの URI を VirusTotal[27] や FireEye[28] のアラートレポートを調べたところ、壁紙カスタマイズアプリケーションの Androidesk_Wallpaper_3_0_9.apk 及び Androidesk_Wallpaper_3_3.apk が発した http://members.3322.org/dyndns/getip はトロイの木馬であることがわかった。一方で他のアプリケーションについては、不信なアプリケーションであるという確固たる証拠を得ることができなかった。しかし、アプリケーションのパッケージ名を APKTOP 及び GooglePlay で調べたところ、同名のアプリケーションを特定できた。その中には有償アプリケーションもあった。このことから APKTOP や OperaMobileStore のアプリケーションには正規品を再パッケージ化して配布したアプリケーションがある可能性がある。

5.2 実験の考察

実験結果から、APKTOP について次のことがわかった。

- apktop crawler でアプリケーションをダウンロードした数日後、再度クローラを実行したと

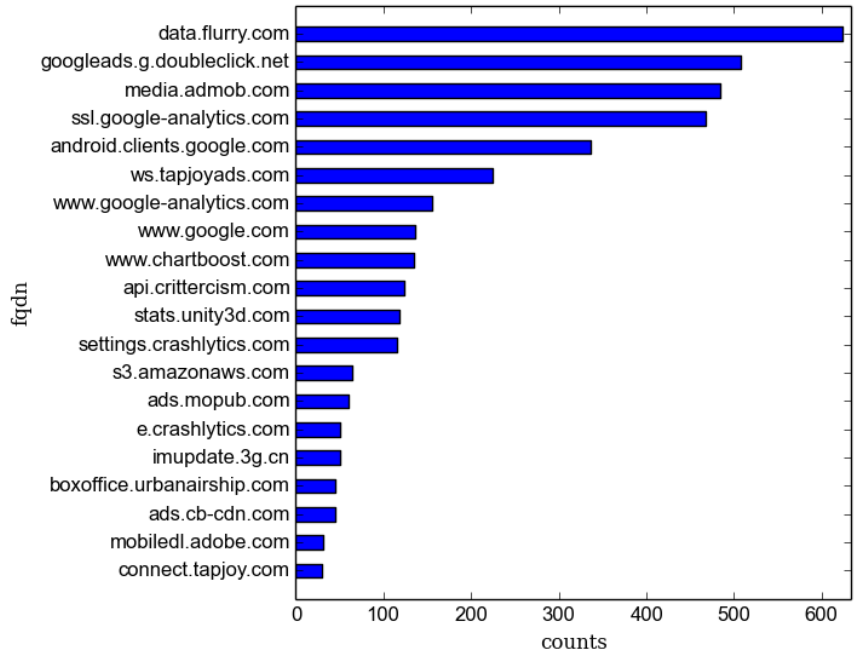

図 5.7: apktop に属するアプリケーションの FQDN と種類

ころ、アプリケーションをダウンロードできた。

- リンク切れのページが多くある。
- 強制終了するアプリケーションが多い。
- 管理者権限を要求するアプリケーションが多い。
- HTTP 通信の多くは広告以外に利用している。
- トロイの木馬を含む HTTP 通信を発するアプリケーションを発見した。
- 同じパッケージ名を持つ有償アプリケーションが GooglePlay にて公開されていた。

この結果から、APKTOP の管理者はいない、もしくは管理を怠っていると考えられる。Web ページのリンク切れや強制終了するアプリケーションを多く抱えていることから、このマーケットの管理者はインターネットで拾ってきたアプリケーションを動作チェック、セキュリティチェックすることなく公開している可能性がある。もしくは管理者は不在で、アプリケーションを公開したい開発者が勝手に公開している可能性もある。また、トロイの木馬を含む HTTP 通信を発するア

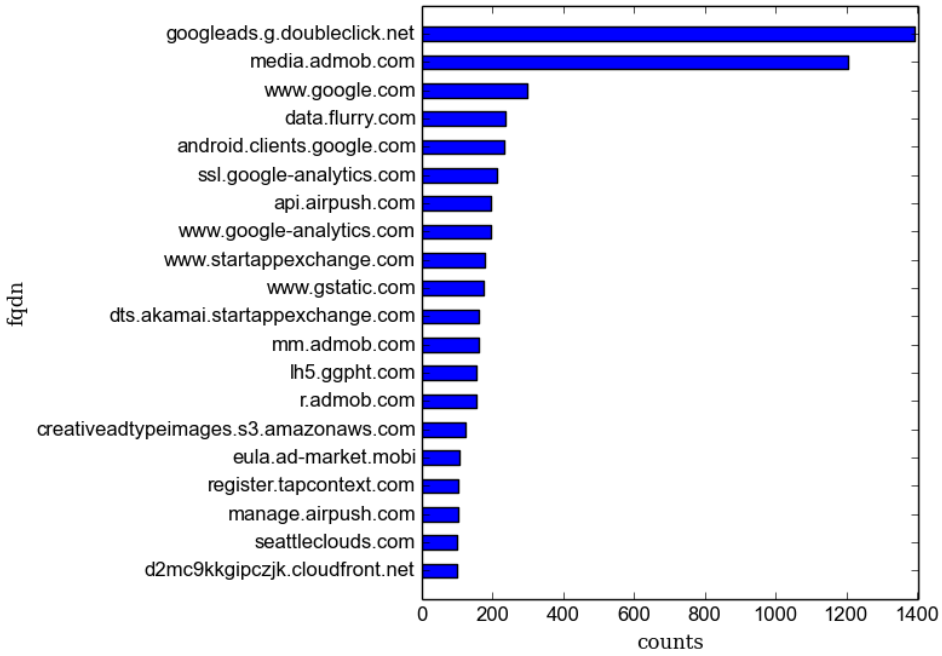

図 5.8: OperaMobileStore に属するアプリケーションの FQDN と種類

プリケーションの発見や、同じパッケージ名を持つ有償アプリケーションが GooglePlay に登録されていたことから、このマーケットを利用することは危険であると考える。

次に、OperaMobileStore について次のことがわかった。

- opera crawler でアプリケーションを集めた数日後、再度クローラを実行したが、アプリケーションをダウンロードすることができなかった。
- HTTP 通信は広告に関するものが多い。それら広告は Adblock にヒットするものが多かった。
- 強制終了するアプリケーションが少ない。
- 管理者権限を要求するアプリケーションがなかった。

この結果から、OperaMobileStore はユーザにとって安全なマーケットである可能性が高いと考えられる。これは、opera crawler を数日後に再実行した際、アプリケーションダウンロードがうまくいかなかった。つまり、対策が施されていたと考える。また、強制終了するアプリケーションが少ないことから、マーケットに公開する際に、動作テストを行っていると考える。最後に管理者権限を要求するアプリケーションがなかったことから、このマーケットは Android 端末の Root 化に対して否定的な姿勢であると考える。けれども、迷惑な広告を含むアプリケーションが多いこと

表 5.11: adchecker の各フィルタのヒット数

フィルタ	APKTOP	OperaMobileStore
General	451	2,912
Japan 1	478	2,354
Japan 2	559	3,123
English 1	267	269
English 2	10	15
Estonia	294	1,790
China	7	75
Russia	8	57
Lithuania	0	1
France	0	1
Malware	4	8

表 5.12: 表 5.11 の項目の説明

フィルタ名	内容
General	不要なフレームや画像、オブジェクトなど英語のウェブページから広告を削除する。[29]
Japan 1	広告とアクセス解析をブロックする。[30]
Japan 2	広告とアクセス解析をブロックする。 [31]
English 1	個人情報を保護するため、トラッキングを削除する。 [32]
English 2	ポップアップをブロックし、Web ページの読み込みを速くする。[33]
Malware	既知のマルウェアもしくはスパイウェアのドメインを検知しブロックする。[34]
国名	各地域ごとの迷惑広告をブロックする。[29]、[35]、[36]

から、OperaMobileStore のユーザは知らずのうちにアドウェアを端末に導入している可能性がある。以上の 2 つのマーケットを比べると、ユーザにとって安全なマーケットは OperaMobileStore である、と考える。

表 5.13: アドウェアを持つ可能性があるアプリケーション

所属マーケット	アプリケーション名
APKTOP	Quick_Launch_Social_Lockscreen_1_30.apk
	Android_Weather_amp_Clock_Widget_3_5_6.apk
	Androidesk_Wallpaper_3_0_9.apk
	Androidesk_Wallpaper_3_3.apk
OperaMobileStore	kairosoft_booklet_ads.apk
	eggplant_marketing.apk
	r8821.apk
	ets_company.apk
	baccaratbaccarabyuyi.apk
	christmas_frame_widget_seventh.apk
	hindi_movies_online_hd_plus.apk

第6章　今後の展開と結論

本章では初めに FireMarker の設計及び実験で明らかになったことを説明し、今後の仕事について述べる。次に本研究の結論を述べる。

6.1　今後の展開

6.1.1　Crawler が扱う Android マーケット

本研究では Android マーケットである APKTOP と OperaMobileStore の 2 つを扱った。今後も調査を継続するためには既存クローラの改善とともに他のマーケットのクローラを開発し、定期的にアプリケーションを集めることが必要である。しかし、Crawler が行うことは言い換えれば Android マーケットへのハッキングである。Nicolas ら [8] も、様々なハッキング技術を利用して GooglePlay からアプリケーションをダウンロードしていると述べている。そのため、何度も対策が取られると考えられる。また最悪、マーケット管理者から通報される可能性もある。従って今後、様々なマーケットからアプリケーションをダウンロードする場合、各マーケットが施しているセキュリテイ対策は大きな障害になると考える。この障害を解決するの今考えられる方法は、2 つある。

1 つ目は、1 度 Android 端末にアプリケーションをインストールしてから、ファイルサーバに転送する手法である。Android のアプリケーションインストールは “/system/app/” フォルダに保存されるので、このフォルダ下にあるファイルを adb pull によってアプリケーションである APK ファイルを取り出すことができる。アプリケーションを一度端末に取り入れてから引き出す手法を複数の端末を用いることで、各マーケットの監視網から逃れられるのではないか、と考えている。2 つ目は、各マーケットの専用アプリケーションの通信をスニッフィングし、アプリケーションを収集する手法である。この通信をスニッフィングする手法は、例えば GooglePlay という名前のアプリケーションの通信を調べ、アプリケーションに該当するパケットを選択し、直接指定したフォルダに保存する。Android アプリケーションは HTTP 通信が多い為、実装できると考えられる。以上 2 つの手法を利用して、各マーケットの専用クローラを開発し、Android マーケットのアプリケーションを観測及び収集を今後の課題とする。

6.1.2 adb の接続障害

5 章で行った FireMarker の実験で、マシンと Android 端末との TCP 通信は約 300 回以上切断したことを確認した。また、これが原因で幾つかのアプリケーションから情報を収集できなかった。アプリケーションから確実に情報を取ること、また実行時間を短くすることから、この障害は無視できない。しかし、ADB の接続障害は既知の問題であるり、数年経った今でも改善はされていない。6.1.2 節ではマシンと Android エミュレータ/デバイスをつなぐツールとして、adb の接続障害をここで説明する。

adb は現在 Android 端末とマシンをつなぐ唯一のツールである。マシンと端末との接続は 2 つ方法があり、USB ケーブルを介した接続と TCP 通信を利用した接続である。どちらかの方法でマシンと端末が接続すると、マシンから Android 端末の接続状態を確認したり、情報を端末内に送ったり、引き出したりできる。その際の結果は標準出力に表示される。adb の各コンポーネントのログを得るには、ADB_TRACE 環境変数を利用する。また、adb bugreport で端末のバグレポートを含む、端末内で起きた全てのログを出力することができる。adb に関するエラーはこれらを用いて調べる。

前述の方法で、設計パターン B で利用した Nexus7 の 1 台から通信障害に関する情報を集めた。結果、adb bugreport からは該当箇所は見つからなかった。tcpdump による結果からは、Nexus7 側から RST を接続マシン側に送信していたことがわかった。しかし、以上の情報からマシンとデバイスの TCP 通信の切断の直接的な原因を得ることができなかった。従って、adb は FireMarker にとって内部に深刻なバグを含んだツールであると言える。接続障害の問題を解決するには、adb ではなく別のツールを開発する、もしくは adb を改善する必要がある。

ここで adb に関する先行開発として、米国 Facebook 社が開発する fb-adb[37] を紹介する。これは adb に実装されていなかった標準エラーを出力したり、adb shell コマンドの status code を返したり、他ツールが標準で実装していることを adb に実装したものである。しかし fb-adb を調べた限りでは、マシンとエミュレータ/デバイスとの接続を安定させる実装は行っていなかった。故に、私が改良する adb は fb-adb を踏襲しつつ、マシンとエミュレータデバイスとの接続を可能な限り安定させ、エラーの原因究明を容易にするものを開発する。

それにはエラー処理のためにプロセスを削除したとしても、マシンと Android 端末との接続は切れない仕組みを考える必要がある。ここで私は adb server を排した adb、zon-adb を提案する。zon-adb は adb server の機能を adb client に移した connect-adb、複数の connect-adb を管理する master-adb、端末内のバックプロセスで実行される adbd から構成される。connect-adb は複数起動することができ、1connect-adb につき 1 台の端末を管理する。connect-adb がどの adbd とも接続していないとき、USB ポートや現在のプロセスをスキャンしてマシンと接続している端末を探す。未接続 adbd を見つけたとき、その adbd と接続を試みる。master-adb は複数の connect-adb を管理する。master-adb が実行されると、現在 connect-adb が何プロセス起動していて、どの端末に接続しているか、

connect-adb のプロセス名とデバイス ID を表示する。また、connect-adb と adbd 間で生じるエラーを出力する。そのエラー出力はなぜ起きたのか、表示し、指定のログファイルに保存する。

この zon-adb が実現できれば、Dispatcher による接続障害が改善され、失敗率を下げることができる。そして、長年放置されていた接続障害の問題が解決され、Android エミュレータ/デバイスの制御やアプリケーション開発がより容易になると私は考えている。

6.1.3 FireMarker で利用する Android 端末

今回、5 章の実験では Android エミュレータは AndroVM を、デバイスは Nexus7 を利用した。今後、さらなる処理時間の短縮や多くのアプリケーションから情報を収集するには、前述した adb の接続問題に加え、利用するエミュレータ/デバイスの数を増やす必要がある。そこで FireMarker はエミュレータか、デバイスか、それとも両方使う方が良いのか、ここで議論する。

初めに Android エミュレータとデバイスの利点欠点を述べる。Android エミュレータの利点は、容易に同品質の端末環境をマシン上に多く用意できることである。加えて新規作成や削除が容易なことから、常にクリーンな環境を提供することができる。Android は情報端末に関係なく動作するように設計されているため、エミュレータで動作したバイナリ・イメージはそのままデバイスで動作する。つまり、エミュレータで取得した情報とデイバスで取得した情報はほぼ同等ともものとなる。この利点により、Dispatcher はアプリケーションから得る情報ときに生じるノイズを小さくし、対象のアプリケーションのみの情報を得ることができる。

しかし、エミュレータはマシンのハードウェア資源を多く消費する。特にグラフィックに関係する資源の消費が激しい。オンボードグラフィックのみで Android エミュレータを 10 数台を起動すれば、すぐさま他のプログラムの処理や 1 つ 1 つのエミュレータの動作に支障がでること確認した。それゆえ、複数の Android エミュレータを利用するには、ある程度の性能を備えたマシンを用意する必要がある。

ここで Android エミュレータを作成できるソフトウェアを紹介する。

Android Virtual Device（AVD）

Android Virtual Device（以下 AVD）は Android SDK の標準ツールの 1 つである。AVD は Android 開発者に各 OS バージョンのエミュレータを容易に作成、提供する。作成直後のネットワーク設定は NAT になっており、外部ネットワークに直接通信するには、幾つかの設定を行う。また AVD は CPU、ハードウェア、DalvikVM と 3 重にエミュレートすることが知られており、動作は軽快ではない。米国 Intel 社が提供している Intel x86 Atom System Image と、Intel x86 Emulator Accelerator（HAXM installer）を用いることで速度向上が望める。エミュレータのバックアップは AVD を構築

しているファイルをバックアップすることで行うことができる。リストアはそれらファイルを操作後のファイルに上書きすれば行える。

Oracle VM VirtualBox

Oracle VM VirtualBox（以下 virtualbox）は米国 Oracle 社が開発している仮想化ソフトウェアパッケージの 1 つである。このソフトウェアを利用して、Android エミュレータを作成することができる。Android の ISO イメージは Android-x86-Porting Android to x86 にて配布されており、イメージを virualbox にインストールすることで利用できる。これで作成された直後のエミュレータのネットワークは NAT になっているが、幾つかの設定を行うことで、外部ネットワークと直接通信することができる。バックアップ/リストアはスナップショット機能によって、容易に行える。

Genymotion（旧:AndroVM）

Genymotion は virtualbox を利用した Android エミュレータ提供サービスである。従来のエミュレータと比べて、起動が早いことが知られている。また様々なメーカがカスタマイズした AndroidOS を利用することができる。無料で利用することもできるが、カメラやデバイス ID、JavaAPI などもエミュレートしたい場合、月々$24.99 を支払うことでエミュレートできるようになる。

Kernel-based Virtual Machine

Kernel-based Virtual Machine（以下 kvm）はイスラエル Qumranet 社が開発したハイパーバイザーで、Linux カーネルに標準で実装されている仮想化技術の 1 つである。kvm で Android エミュレータを起動する場合、起動時に幾つかのオプションをつけて実行することで、様々な Android 環境を実現することができる。バックアップ/リストアはデフォルトの AndroidOS を入れた img ファイルを作成し、それを置き換えることで容易に実現できる。

Android エミュレータをビルドして利用する方法

最後に、Android のソースコードを米国 Googles 社のページからダウンロードして、Android エミュレータそのものをビルドして利用する方法がある。実際にビルドしたところ、ネットワーク接続の方法や CPU 等、細かい設定を施して起動できる。バックアップはエミュレータを構成するファイルをコピーすることで得られる。リストアは上書きすることで実現できる。TaintDroid のような Linux カーネル層に独自のコードを組み込んだ解析ツールを利用する場合は、今の所、これを利用する以外に方法がない。

一方、Android デバイスは、電話応答ができることやマシンのハードウェア資源を消費しない、エミュレータよりも動作が早い。電源が消耗することを解決できれば、ユーザが利用する環境を再現できる。他には、どの仮想化技術にも干渉しない点が挙げられる。しかし、デバイスは数を揃えるのが難しい。各メーカが AndroidOS に一定のカスタマイズを施しているので、実験で利用する場合、デバイスの種類を揃える必要がある。またバックアップには別の専用ソフトウェアを利用しなければならず、容易に OS をクリーンにすることはできない。といった問題がある。

Android デバイスを Cleansing し、クリーンな環境にするには次の方法がある。

アプリケーションをアンインストールする。

アプリケーションをアンインストールすると、端末内にある APK ファイルが削除される。しかし、APK ファイルが設定したファイルや保存フォルダ等が残ってしまう。またアンインストールが失敗すると、APK ファイルが残留してしまう。ゆえに Cleansing の効果は低いと言える。この状態からアプリケーションを削除するには再度アンインストールを試みるか、端末を初期化する。

端末の再起動を行う。

端末を再起動すると、今までに実行していたプロセスが削除される。例えば、Nexus7 とマシンとの接続が切断されたときに復旧する方法として有効である。しかし、Android マルウェアが Nexus7 内に侵入していた場合、マルウェアは再起動する。ゆえに Cleansing の効果は低いと言える。マルウェアを削除するには、端末の初期化が必要である。

デバイスの初期化を行う（工場出荷状態にする）。

この操作は、Nexus7 の「設定」アプリケーションから「バックアップとリセット」へ移動した後、「情報の初期化」ボタンを押すと実行する。この操作によって、アカウント削除、インストールした APK ファイルの削除、各設定のリセット、リカバリー領域削除が行われる。しかし、Root 権限で導入した APK ファイルや写真といった画像ファイルは残留する。これらを全て削除するには OS を再インストールする必要がある。ゆえに Cleansing の効果は中程度と言える。

OS を再インストールをする。

この操作によって、Nexus7 内の設定やアプリケーション、残留ファイル等が完全に削除される。また管理者権限を取得していた場合、その権限も放棄する。ゆえに Cleansing の効果は高いと言える。しかし、再度管理者権限を取得しようとするには、初期設定をした後、管理者権限を取得する操作を手動で行う必要がある。

バックアップからリストアする。

Nexus7 に限らず、Android 端末をバックアップとリストアするには数種方法がある。ここでは管理者権限を取得した Nexus7 のバックアップとリストアについて述べる。管理者権限を取得した端末に限り、全ての情報をバックアップ・リストアすることができる。その代表的なものとして、ClockworkMod や TWRP といったカスタムリカバリツールがある。

これらツールは Nexus7 のリカバリ領域に上書きすることで利用することができる。また、userdata 領域や system 領域などをフルバックアップ/リストアできる機能を持つ。カスタムリカバリツールを利用し、初期状態をバックアップ、端末操作後にリストアすることで、初期状態を保つことができる。ゆえに Cleansing の効果は高いと言える。しかし、これらのツールは外部コマンド（例えば、adb や fastboot コマンドなど）で操作することができない。この機能をシステムに取り入れる場合は、リカバリ領域で利用できるツールの開発か、別の方法によって実現する。列挙した通り、デバイス上の環境をクリーンにするには、多くの手順を踏む。以上のことから、アプリケーションから情報を収集するにはエミュレータを用いた方が望ましい、と考える。

最後に今後としての FireMarker の MarketDrone、Dispatcher の改善案について述べる。Android エミュレータをシステムに組み込むには、次の条件があると考える。GUI を利用せず CUI で操作可能であること最低条件であると考える。

1. CUI で利用可能である。
2. 起動時に必要な設定を施せる。
3. エミュレータの維持に必要なハードウェア資源の消費を抑える。

まず CUI による操作であるが、Android エミュレータは何れも GPU を使いグラフィカルな操作を要求する。しかし、一部のエミュレータソフトウェアはバックプロセスで動作でき、ディスプレイ画面に表示させずに処理を行える。ハードウェア資源の消費を抑えるため、バックグラウンドプロセスで起動できるエミュレータが望ましいと考える。

この点を満たすエミュレータは、調べたところ AVD と virtualbox、Android エミュレータをビルドして利用する方法が妥当であるとわかった。起動時に必要な設定を施せる観点から言うと、virtualbox はできない。しかし、軽快さで言えば virtualbox に勝るエミュレータはなかった。virtualbox の場合は必要分だけ手動で設定し、スナップショット機能を用いてバックアップを取り、利用する。デフォルトのエミュレータを用いて情報収集するには virtualbox を利用し、カスタム OS を利用する場合は Android エミュレータをビルドしたものを用いるとバックアップ/リストアによる Cleansing が容易にできることがわかった。これらのソフトウェアの制御を Dispatcher に組み込むことを今後の課題とする。

6.1.4 MarketDrone の調査項目の追加

5 章の実験において、システムログからは強制終了と管理者権限を要求するアプリケーションを特定した。HTTP 通信からは Adblock のブラックリストに該当する迷惑広告を持つアプリケーションを特定した。マーケットをより多角的に見るため、解析項目を増やすことは重要である。その候補として、Google アカウント利用の要求、アプリケーションが通信する Web サーバの実態である。Google アカウント利用の要求は、McAfee のレポート [3] からヒントを得た。

レポートによれば、アプリケーションがユーザから Google サービスへのアクセスを許可すると、認証トークンを不正に利用し、GooglePlay サーバに接続、別のアプリケーションをパーミッション機構を通さずに自動インストールして起動する Android/Badlnst.A を発見した、と述べている。その他にも、ユーザに知られることなく Google アカウント ID や IMEI（携帯端末識別番号）、IMSI（携帯電話加入者識別番号）を特定の Web サーバに送信するアプリケーションも見つかっている。

このことから、アプリケーションのマニフェストファイルを解析、Google アカウントサービスの要求に関するパーミッションを探すとともに、アプリケーション実行時の通信を監視することが必要である。

次にアプリケーションが通信する Web サーバの実態を明らかにし、それが不審なサーバか否か明らかにすることである。adchecker を用いた解析から、旧共産圏に該当する国々のフィルタにヒットするアプリケーションを特定した。特にエストニアは歴史的な背景として、2007 年 4 月に電子投票システムを狙った大規模なサイバー攻撃に晒された国である。

この事実から、アプリケーションが通信する Web サーバの実態を明らかにすることは利用リスクを考える上で有意義であり、アプリケーション評価項目の 1 つとなりうると考えている。そのほか、2.2 章で紹介した既存動的解析手法を調査項目の 1 つとして利用しようと考えている。

6.1.5 FireMarking の設計と実装

5 章で行った実験で、APKTOP の OperaMobileStore の一傾向を明らかにした。結果、APKTOP は配布サイトとして整備されておらず、利用するリスクは高い可能性を示し、OperaMobileStore は管理者が定期的に整備しており、APKTOP と比べ、安全である可能性を示した。次に FireMarking を設計及び実装するため、次のことを順に行う。

1. Android マルウェアの調査による信用できない Android マーケットの基準の決定
2. 5 章の Filter の実験で得られた結果の解析

初めに、MarketDrone を用いて Android マルウェアから情報を得る。対象となるマルウェアは contagion というマルウェアを配布している一般 Web サイト [13] からダウンロードできるものすべてとし、この Web サイトを Android マルウェアマーケットと定義する。次に実験で得た MarketDrone

で APKTOP、OperaMobileStore の結果とマルウェアマーケットから得た結果から、それぞれの分布を調べる。もしくは自己組織化マップを用いて、3 つの結果を比較できるようにする。APKTOP、OpreaMobileStore の分布もしくは自己組織化マップの結果がマルウェアマーケットの結果と似ていれば似ているほど、マーケット利用リスクは高いとする。最後に前述までのマーケット解析結果と 2.2 節で紹介した既存の動的解析手法の出力、セキュリティ企業が報告するブラックリストなどを用いて、対象マーケットの各アプリケーションの評価を測定する。この一連の試行と実装を今後の課題とする。

6.2 結論

Android 端末の急速な普及に伴い、Android マルウェアによる被害報告が増えている。その原因は 3 つのセキュリティレポートから、2 つあると考えた。それは、ユーザが Android アプリケーションを十分な情報に基づいた意思決定によって、アプリケーションをインストールしているわけではないこと、Android マーケット運営者によって、マーケットは適切に管理されているわけではないことである。

そこで本研究では、第 3 者の視点から Android マーケットを分析、評価するシステム、FireMaker を提案した。提案システムは、複数のマシンと複数の Android 端末を制御して、評価対象となる Android マーケットのアプリケーションを自動的に操作して情報を得る。その情報から、マーケットの傾向分析を行う。さらに、分析結果からマーケットの運営状況や利用リスク、各アプリケーションの利用リスクをユーザに示す。FireMarker によって、調査した各マーケットの情報をデータベースに保存し、誰でもアクセスするにようになれば、Binarian は提案する手法の検証の際、データセットとして利用することができるようになった。また、各マーケットの情報を統計学的に分析した結果を公開すれば、Analyst は各マーケットの動向を容易に理解することが可能となった。さらに、End-user は FireMarker によって、アプリケーションインストール時に利用するマーケットの安全性を確かめることができる。その結果、End-user はマーケットの安全性を考慮した意思決定により、アプリケーションを使うか否か決めることができる。

本研究では提案システムのうち、複数のマシンと複数の Android 端末を制御して、マーケットにあるアプリケーションを自動的に操作し情報を得るシステムを実装した。そのシステムの実証実験から、2 つのマーケットの傾向を明らかにした。さらに、傾向分析を行い、各マーケットの潜在的なリスクの有無について示した。結果から、2 つのマーケットを比較し、どちらがユーザにとって安全なマーケットであるか示した。最後に今後の展開として、その情報を利用した各マーケットに属するアプリケーションを評価する手法を示した。

第7章　謝辞

本研究を行うにあたり、多くの方から多大なご助言やご助力をいただきました。それらの方々のご協力がなければ、本研究は成り立ちませんでした。深く感謝し、心から厚くお礼申し上げます。

本研究を進めるにあたり、指導教員である篠田陽一教授には様々な助言、適切なご指導賜りました。心から深く感謝致します。また、助言を頂いた副指導教員である丹康雄教授、副テーマ指導教員である面和成准教授に感謝致します。

本研究室の知念賢一特任准教授には研究に関して活発な議論や多大なご指導を賜りました。心から感謝致します。

情報通信研究機構の高野祐輝博士、太田悟史氏、三浦良介氏には研究に関して様々な助言、適切なご指導賜りました。心から感謝致します。

篠田研究室の宇多仁助教、井上朋哉助教、Muhammad Imran Tariq 氏、明石邦夫氏、岩橋紘司氏、成田佳介氏、岩本裕真氏、園田真人氏、八木辰弥氏、可児友邦氏には研究以外でも普段の生活の面で支えていただきました。 ここに心から感謝致します。

最後に研究や生活で支えてくれた家族へ心から感謝します。

参考文献

[1] Android Developers. http://developer.android.com/index.html.

[2] GData Mobile Malware Report Half-yearly report January June 2013. https://blog.gdatasoftware.com/uploads/media/GData_MobileMWR_H1_2013_EN_01.pdf.

[3] McAfee Labs Threats Report. http://www.mcafee.com/jp/resources/reports/rp-quarterly-threat-q1-2014.pdf.

[4] ノートン モバイルアプリ調査. http://jp.norton.com/now/ja/pu/images/Promotions/2015/NMS/mobileapp_survey.pdf.

[5] GooglePlay. https://play.google.com/store.

[6] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, Phillipa Gill, and David Lie. Short paper: A look at smartphone permission models. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pp. 63–68, New York, NY, USA, 2011. ACM.

[7] ADB(Android Debug Bridge): How it works? http://www.slideshare.net/tetsu.koba/adbandroid-debug-bridge-how-it-works.

[8] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. *SIGMETRICS Perform. Eval. Rev.*, Vol. 42, No. 1, pp. 221–233, June 2014.

[9] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*. The Internet Society, 2012.

[10] Wei Xu, Fangfang Zhang, and Sencun Zhu. Permlyzer: Analyzing permission usage in android applications. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, pp. 400–410. IEEE, 2013.

[11] Jonathan Crussell, Ryan Stevens, and Hao Chen. Madfraud: Investigating ad fraud in android applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pp. 123–134, New York, NY, USA, 2014. ACM.

[12] 隆将磯原, 秀明川端, 敬祐竹森, 歩窪田, 潤也可児, 晴信上松, 正勝西垣. セカンドアプリ内包型 android マルウェアの検知. コンピュータセキュリティシンポジウム 2011 論文集, 第 2011 巻, pp. 708–713, oct 2011.

[13] contagio mobile. http://contagiodump.blogspot.com/.

[14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. p. 5, 2014.

[15] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pp. 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[16] 弘樹葛野. Android アプリケーションに対する情報フロー制御機構の提案. コンピュータセキュリティシンポジウム 2011 論文集, 第 2011 巻, pp. 155–160, oct 2011.

[17] Google Play デベロッパー プログラム ポリシー. https://play.google.com/intl/ALL_jp/about/developer-content-policy.html#showlanguages.

[18] tcpdump. http://www.tcpdump.org/.

[19] APKTOP. http://www.papktop.com/.

[20] Adblock. https://adblockplus.org/.

[21] jailbreak. http://en.wikipedia.org/wiki/IOS_jailbreaking.

[22] android developers blog -It 's not" rooting ", it 's openness-. http://android-developers.blogspot.jp/2010/12/its-not-rooting-its-openness.html#uds-search-results.

[23] Rooting (Android OS). http://en.wikipedia.org/wiki/Rooting_(Android_OS).

[24] Privoxy. http://www.privoxy.org/.

[25] Genymotion. http://www.genymotion.com/blog/.

[26] Opera Mobile Store. http://apps.opera.com/_jp/.

[27] VirusTotal. https://www.virustotal.com/ja/.

[28] FireEye. https://www.fireeye.com/jp/ja/.

[29] EasyList. https://easylist.adblockplus.org/en/.

[30] 豆腐フィルタ. http://tofukko.r.ribbon.to/abp.html.

[31] adblock-plus-japanese-filter. https://code.google.com/p/adblock-plus-japanese-filter/.

[32] anboy's Annoyance List. https://easylist-downloads.adblockplus.org/fanboy-annoyance.txt.

[33] EasyPrivacy. https://easylist-downloads.adblockplus.org/easyprivacy.txt.

[34] MalwareDomains. \http://www.malwaredomains.com/?page_id=2.

[35] ruadlist. https://code.google.com/p/ruadlist/.

[36] Bulgarian list. http://stanev.org/abp/adblock_bg.txt.

[37] fb-adb. https://github.com/facebook/fb-adb.