情報処理学会論文誌 Vol.56 No.12 2275–2288 (Dec. 2015)

Androidアプリケーションの自動リパッケージに対する耐性評価

金井 文宏[1,†1] 庄田 祐樹[1] 橋田 啓佑[1,a)] 吉岡 克成[2] 松本 勉[2]

受付日 2015年3月9日, 採録日 2015年9月2日

概要：スマートフォン向けOSとしてAndroidが広く用いられている一方で，それを狙ったマルウェアの数も増加している．Androidマルウェアの中には，リパッケージと呼ばれる手法を用いて，正規アプリの中に悪性コードを追加することで作成されたものが多く存在する．攻撃者がリパッケージマルウェアを大量に作成する際には，リパッケージ処理の自動化が必須であると考えられるが，自動リパッケージの実態や対策については，十分な調査・検討が行われていない．そこで我々は，既存の正規アプリが自動リパッケージに対して，どの程度の耐性を有するかを検証する．まず，実際のリパッケージマルウェアの解析を行うことで，リパッケージの方法を特定し，自動リパッケージを再現するスクリプトを作成する．次に，このスクリプトによって，複数の正規アプリに対して，外部と通信を行う機能だけを持つ検証用コードを挿入する．作成したリパッケージ済みアプリを動的解析して，挿入した検証用コードが正常に動作するかどうかを検証する．その結果，自動リパッケージの手法により成功率に差がみられるものの，評価対象としたアプリの7〜9割において，挿入した検証用コードが正常に動作し，なおかつ起動時の動作が変化しないことを示す．さらに，ユーザによるインストール数が5,000万件を超える33種類のアプリにおいて，アプリの持つ基本的な機能がリパッケージ後にも保持されるかを確認し，87.9%にあたる29種類のアプリにおいて機能が保持されていることを示す．この実験において挿入した検証用コードを，悪性のコードに変更した場合でも，同様の方法で自動リパッケージが可能であることが予想される．以上より，現状のAndroidアプリの多くは自動リパッケージへの耐性が不十分であり，耐タンパ技術などを用いたリパッケージ対策が必要であることが分かる．

キーワード：Android，リパッケージ

Evaluating Resistance of Android Applications to Automated Repackaging

Fumihiro Kanei[1,†1] Yuki Shoda[1] Keisuke Hashida[1,a)]
Katsunari Yoshioka[2] Tsutomu Matsumoto[2]

Received: March 9, 2015, Accepted: September 2, 2015

Abstract: Android is widely used as a smartphone OS. On the other hand, malware targeting the Android devices is increasing. Sometimes, attackers insert malicious code to benign application to create malware. This technique is called repackaging. Attackers may automate the process of repackaging when they insert malicious code to a large amount of apps. However, difficulty and cost of automated repackaging have not been well-investigated. In this paper, we evaluate resistance of Android apps to automated repackaging. For reproduction of automated repackaging, we insert a test code to benign apps by methods which have been used in actual malware created by repackaging. After repackaging, we check whether the inserted code properly works or not by dynamic analysis. As a result of the experiment, we successfully insert the test code into 75–90% of all tested popular apps without influencing startup screen display of the original apps. In addition, we confirmed that original function is remained after repackaging in 29 out of 33 applications, which was installed by user over fifty million times. As there is no technical difficulty to replace the test code to malicious code, we conclude that many Android apps are lack of resistance to automated repackaging and must consider measures such as tamper resistant software technology.

Keywords: Android, repackaging

1. はじめに

近年，スマートフォン向けOSとしてAndroidが広く用いられている一方で，Androidを狙ったマルウェアの数も増加している．Androidマルウェアには，すでにパッケージ化されたアプリに対して新たにコードを挿入する，リパッケージと呼ばれる手法を利用し，正規のアプリに対して悪性コードを挿入することで作成されたものがある．これらはリパッケージマルウェアと呼ばれる．リパッケージマルウェアは，悪性コード挿入前の元の正規アプリの機能も保持したうえで，多くの場合，サードパーティマーケット上で拡散される．攻撃者は公式マーケットであるGooglePlay上の人気の正規アプリに対してリパッケージを行い，アプリ審査などのセキュリティ機構が不十分なサードパーティマーケット上にアップロードすることで，被害を拡大させていると考えられる．リパッケージマルウェアの機能は，Bot化，RAT化のようなものに加え，典型的な情報摂取機能も備わっている場合も多く，感染ユーザのプライバシへの被害が懸念される．リパッケージマルウェアの被害事例としては，“Android.Troj.mdk” というファミリ名のマルウェアによる事例 [1] があげられる．このマルウェアは，中国のサードパーティ製マーケットで配布されたリパッケージマルウェアであり，7,000種類以上の正規アプリに対してリパッケージが行われ，最大100万台のデバイスに感染したとされている．

上記の事例のように，攻撃者が大量のリパッケージマルウェアを作成する際には，その処理を自動化していることが予想されるが，自動リパッケージの実態やその対策については，十分な検討が行われていない．

そこで我々は，既存の正規アプリが自動リパッケージに対してどの程度の耐性を有するかを検証するための実験を行った．まず，実際のリパッケージマルウェアの解析を行うことでリパッケージの方法を特定し，自動リパッケージを再現するスクリプトを作成した．次に，Web上から収集した複数の正規アプリに対して，外部と通信を行う機能だけを持つ検証用のコードを挿入することで，簡易的に自動リパッケージを行った．さらに，作成したリパッケージ済みアプリを動的解析することで，挿入した検証用コードが正常に動作するかどうかを検証した．その結果，自動リパッケージの際に用いた手法によって成功率に差がみられたものの，元にした正規アプリのうちの約75%～90%において，挿入した検証用コードが正常に動作し，なおかつ元の正規アプリの起動時の動作に変化が生じないことを確認した．さらに，インストール数の観点から，リパッケージが行われることによって大きな被害をもたらすと考えられる33種類のアプリにおいて，アプリの持つ基本的な機能がリパッケージ後にも保持されるかを確認し，87.9%にあたる29種類のアプリにおいて機能が保持されていることを確認した．本実験においては，単純な機能のみを持つ，非悪性の検証用コードを挿入したが，これを実際に攻撃者が利用するような悪性のものに変更した場合でも，同様の方法で自動リパッケージが可能であることが予想される．以上より，現状のAndroidアプリの多くは自動リパッケージへの耐性が不十分であり，耐タンパ技術などを用いたリパッケージ対策が必要であることが分かる．

本論文ではまず，2章でAndroidアプリケーションのリパッケージに関する先行研究について説明する．3章ではAndroidアプリの構造と，リパッケージの手法について述べる．4章でAndroidアプリの自動リパッケージに対する耐性評価実験について述べ，5章で考察を行う．最後に6章でまとめと今後の課題を述べる．

2. 関連研究

Androidアプリのリパッケージに関する先行研究としては文献 [2], [3] などがあげられる．

文献 [2] では，Fuzzy Hashingの技術を用いたDroidMOSSと呼ばれるシステムを構築し，公式マーケットから収集した正規アプリのセットを，サードパーティマーケット内のアプリと比較することで，サードパーティマーケット内のリパッケージされたアプリを検出する手法が提案されている．また，サードパーティマーケットから収集したアプリのうち，5%～13%のものがリパッケージによって作成されたとしており，リパッケージの主な内容としては，広告モジュールの追加・改変や悪意のあるペイロードの追加が見られたとされている．

また，文献 [3] では，既存のリパッケージ検出アルゴリズムが難読化によって大量のFalse Negativeを生じる可能性を指摘し，難読化に対するそれらのアルゴリズムの耐性を評価するためのフレームワークを提案している．

上記のように，リパッケージされたアプリの検知といった観点からの先行研究はいくつか存在するが，既存の正規アプリのリパッケージに対する耐性評価といった観点からのものは少なく，特に自動的なリパッケージについての検証は不十分であるといえる．

[1] 横浜国立大学
Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
[2] 横浜国立大学大学院環境情報研究院/横浜国立大学先端科学高等研究院
Graduate School of Environment and Information Sciences, Yokohama National University/Institute of Advanced Sciences, Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
[†1] 現在，NTTセキュアプラットフォーム研究所
Presently with NTT Secure Platform Laboratories
[a)] hashida-keisuke-dk@ynu.jp

3. Android アプリの構造とリパッケージの手法

本章では，Android アプリの構造，およびリバースエンジニアリングの手法について説明し，攻撃者によるリパッケージの概要について述べる．

3.1 Android アプリの構造

Android アプリは，apk ファイルという zip 形式で圧縮された専用ファイルとして配布されている．apk ファイルの主な中身としては，アプリのメタ情報をまとめた AndroidManifest.xml，Android 上で実行されるバイトコードの本体である classes.dex，画像や音声などのリソースファイル，署名情報が保存されている META-INF フォルダなどがあげられる．それ以外にも，ネイティブライブラリを含んだ lib フォルダや，アプリ作成者が任意のファイルを置くことができる assets フォルダなどが存在する場合もある．

Android アプリは，基本的に Java によって開発が行われ，Java のソースコードが Java バイトコードにコンパイルされた後に，Android 上で動作する Dalvik VM 用のバイトコードである dex 形式のファイルへと変換が行われる．AndroidManifest.xml 内に記述される情報としては，端末内でアプリを一意に定める ID であるパッケージ名，利用するパーミッション，アプリケーションに含まれているアクティビティ，サービス，コンテントプロバイダなどのコンポーネント，それぞれのコンポーネントが実装されているクラス名などがあげられる．また，Android におけるアプリ間連携機能であるインテント [4] に関して，それぞれのコンポーネントが受信するインテントの種類が，インテントフィルタとして AndroidManifest.xml 内に記述される．

3.2 Android アプリのリバースエンジニアリング

前述のとおり，Android アプリの多くは Java によって記述されており，他の言語と比較して，リバースエンジニアリングが容易であるという特徴がある．具体的には，apktool [5] や smali/baksmali [6] などのツールを用いることで，バイトコードである classes.dex を可読であるテキスト形式のファイルにディスアセンブルすることが可能である．**図 1** は apktool のディスアセンブルによって出力されるファイルの構成の一例を示しており，/smali フォルダ以下に含まれている smali ファイルがテキスト形式にディスアセンブルされたコードにあたる．apktool によって出力された各ファイルに対して，コードの改変やリソースファイルの追加を行った後に，再び apktool を用いて apk ファイルへとビルドすることも可能である．再ビルド後には jarsigner [7] コマンドを用いて apk ファイルに対して署名付与を行うことで，再び Android 端末へのインストールが可能となる．

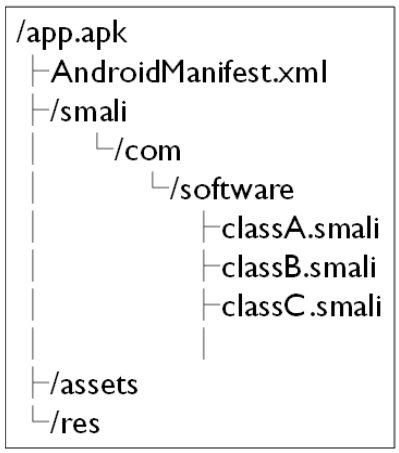

図 1 apktool のディスアセンブルによって出力されるファイルの構成例
Fig. 1 Example of output files disassembled by apktool.

3.3 Android アプリのリパッケージ

攻撃者がリパッケージを行う際には，前節で述べたようなリバースエンジニアリングの技術を用いて，正規アプリをディスアセンブルした後に，悪性コードの挿入，再ビルドを行っていると予想される．

悪性コードの挿入に関しては，挿入されるコードの持つ機能によって様々なパターンが考えられる．たとえば，攻撃者が悪性コードを含んだ Java パッケージ（以下，悪性パッケージ）を作成し，正規アプリに挿入する場合を考えると，挿入される悪性パッケージ内で，元の正規アプリでは利用されていなかったパーミッションやコンポーネントが利用されていた場合，AndroidManifest.xml 内に，それらを定義する記述を追加する必要がある．また，悪性コードのエントリポイントとして，元の正規アプリのコード内に，挿入された悪性コードを呼び出す処理を追加する場合なども考えられる．

上記の例のように，攻撃者によって悪性コードが挿入される際には，実際に悪性な処理を行うコードやパッケージを挿入するのに加えて，AndroidManifest.xml の改変や，元の正規アプリに含まれていたコードの改変を行うことが予想される．

3.4 リパッケージの自動化

自動的なリパッケージの場合を考えると，個々のアプリの内部構造に応じたコードの挿入手法を自動化するのはコストが高いため，機械的に解析が可能な情報（AndroidManifest.xml に含まれる情報や，アプリの Java パッケージの構造など）を利用して多くのアプリに対して共通の処理でリパッケージが可能な手法が用いられると考えられる．これらの制約を考慮すると，挿入される悪性コードの

呼び出され方や正規アプリ内で改ざんされる箇所などに傾向が現れることが予想される．

4. Androidアプリの自動リパッケージに対する耐性評価

本章では，Androidアプリの自動リパッケージに対する耐性評価実験に関して，実験の目的，実験内容，実験結果について述べる．

4.1 実験目的

攻撃者が大量のリパッケージマルウェアを作成する際には，前章で述べたようなリパッケージの手法を自動化していることが予想される．しかし，実際に自動リパッケージを行う際の，リパッケージの成功率や，自動化にかかるコストなどについては，これまで十分に検証が行われていない．また，リパッケージによって生成されたマルウェアは，元の正規アプリの機能を保持しているため，ユーザに継続的に利用され，長期間潜伏する恐れがあることから，その影響は大きいといえる．

そこで本実験では，既存の正規アプリに対して，実際のリパッケージマルウェアにおいて用いられている自動リパッケージ手法により検証用コードを自動挿入し，(1) 挿入した検証用コードが正常に動作するかどうか，(2) リパッケージ済みアプリの動作が元のアプリから変化するかどうか，という2点についての検証を行うことを目的とする．

4.2 実験内容

4.2.1 評価対象の正規アプリ

ダウンロード数などの観点で人気が高いことが予想される1,612種類の正規アプリを2015年2月にWeb上から収集し，これを対象として自動リパッケージの耐性評価実験を行った．**表 1** は収集した正規アプリが利用していたパーミッションの中から，利用される頻度が高かったものの上位10個を抜き出したものである．表の上位2つのパーミッションはアプリがインターネットに接続する際に要求されるパーミッションであり，今回評価対象とした正規アプリのほとんどが，このパーミッションを利用していたことが分かる．

4.2.2 リパッケージマルウェアの解析

実際のマルウェアにおけるリパッケージ手法を調査するため，過去に流行した3種類のリパッケージマルウェアに対して静的解析を行った（**表 2**）．

解析の結果，すべての検体において，悪性パッケージの追加，AndroidManifest.xmlの改変が見られた．また，検体No.1においては，リソースファイルの追加や，正規アプリに含まれていたコードに対する改変も見られた．一方で，それぞれの検体において，追加された悪性コードの呼び出され方が異なっていた．**図 2** に各検体における悪性パッケージが呼び出されるまでのフローを示す．検体No.1においては，挿入された悪性コードは正規アプリのアクティビティの中から呼び出されていたのに対して，検体No.2においては，“android.intent.action.MAIN” のインテントによって悪性コードが呼び出され，その後悪性コード内で元の正規アプリのアクティビティを明示的インテントによって呼び

表 1 評価対象の正規アプリが利用するパーミッション（上位10位まで）
Table 1 Permissions used by evaluated apps (Top 10 permissons).

パーミッション名	アプリ数
android.permission.INTERNET	1545
android.permission.ACCESS_NETWORK_STATE	1445
android.permission.WRITE_EXTERNAL_STORAGE	1087
android.permission.WAKE_LOCK	898
com.google.android.c2dm.permission.RECEIVE	692
android.permission.GET_ACCOUNTS	686
android.permission.VIBRATE	635
android.permission.ACCESS_WIFI_STATE	623
android.permission.READ_PHONE_STATE	611
com.android.vending.BILLING	424

表 2 静的解析対象のリパッケージマルウェア
Table 2 Target malwares created by repackaging.

検体No	検知名（AVベンダ名）	MD5
No.1	Andr/Ksapp-A (Sophos)	97885f41713e7cc888567438626b0711
No.2	Android.Geinimi (Symantec)	90c05039fd16b78a7d4e7aaf803afaaa
No.3	Android/AndroRAT (AVG)	f64f38cdd6d12f10b1f9ee4bfb46ffc9

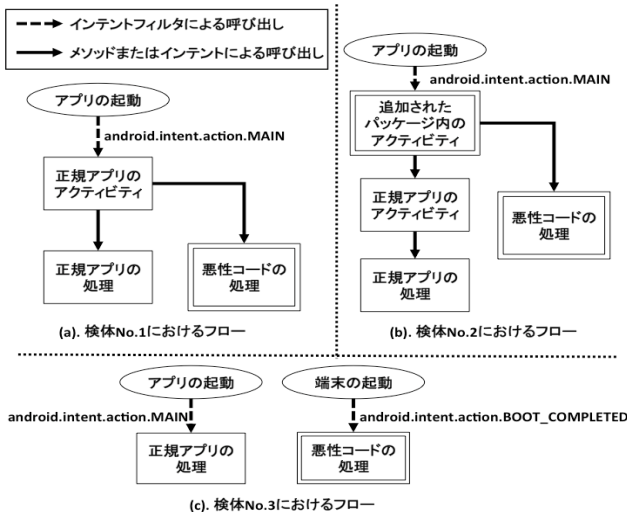

図 2 検体No.1，2，3における悪性コードが呼び出されるフロー
Fig. 2 Flowchart of malicious code execution (Sample No.1, 2, 3).

出していた．また，検体 No.3 においては，アプリの起動時には悪性コードが呼び出されず，端末が起動した際に発行される “android.intent.action.BOOT_COMPLETED” のインテントによって悪性コードが呼び出されていた．

上記の内容から，それぞれの検体において以下のような手法で悪性コードの挿入が行われたことが予想される．

手法 1（検体 No.1）

① 悪性パッケージの追加．
② AndroidManifest.xml 内にパーミッション，コンポーネントを追加．
③ “android.intent.action.MAIN” のインテントフィルタが設定されているアクティビティを特定．
④ 該当アクティビティの onCreate メソッドの先頭部分に悪性パッケージ内のコードを呼び出すためのコードを挿入．

手法 2（検体 No.2）

① 悪性パッケージの追加．
② AndroidManifest.xml 内にパーミッション，コンポーネントを追加．
③ “android.intent.action.MAIN” のインテントフィルタが設定されているアクティビティを特定．
④ 該当アクティビティのインテントフィルタを削除．
⑤ “android.intent.action.MAIN” のインテントフィルタを，追加した悪性パッケージ内のアクティビティに設定．
⑥ 悪性コード内に，手順 ③ において特定したアクティビティを明示的インテントによって呼び出す処理を追加．

手法 3（検体 No.3）

① 悪性パッケージの追加．
② AndroidManifest.xml 内にパーミッション，コンポーネント（android.intent.action.BOOT_COMPLETED のインテントフィルタを設定済み）を追加．

4.2.3 自動リパッケージ

以下の手順で自動リパッケージを行うためのスクリプトを作成した．

1. **apktool を用いて自動リパッケージ対象の正規アプリをディスアセンブル**
2. **検証用コードの挿入**
3. **apktool を用いて再ビルド**
4. **jarsigner コマンドによる署名付与**

今回の実験においては，手順 1，3 における正規アプリのディスアセンブル・再ビルドの際には apktool（version 2.0.0RC3）を用いた．また，手順 2 においては挿入する検証用コードとして外部に用意したサーバと通信を行う機能を持つコードを用意した．検証用コードの挿入は，前項において解説した 3 種類のリパッケージマルウェアによって用いられていた手法（手法 1，2，3）と同様の手順で行った．手法 1 において，onCreate メソッドの先頭部分に悪性パッケージ内のコードを呼び出すためのコードを挿入する際には，apktool によって出力された smali ファイルに対する文字列マッチングを用いた．具体的には起動時に呼び出されるアクティビティが実装されているクラスの smali コードの中で “->onCreate(Landroid/os/Bundle;)” の文字列が発見された場合，その次の行に悪性パッケージ内のコードを呼び出すための smali コードを挿入した．さらに，手順 4 における jarsigner コマンドによる署名付与の際には，独自に生成した署名生成鍵を用いた．

4.2.4 リパッケージ済み正規アプリの動的解析

リパッケージ元のアプリと，各手法によってリパッケージされたリパッケージ済みアプリを動的解析することにより，挿入した検証用コードが動作するかどうか，また，リパッケージによって元の正規アプリの機能に変化が生じていないかどうかを検証した．

本実験で用いた動的解析の環境は以下のとおりである．

- 動的解析環境

端末：Galaxy Nexus GT-I9250
OS：Android OS 4.2.1
ネットワーク：WiFi でインターネットへ接続

今回の実験においては，動的解析を行う環境として，Android OS 4.2.1 を搭載した実機を用いた．本実験で用いた自動リパッケージの手法は，OS のバージョンに依存しないため，異なるバージョンを用いた場合でも，本質的な実験結果は大きく変化しないと考えられるが，今後の Android OS のバージョンアップにおいて，フレームワークの仕様に大きな変化があった場合はこの限りではない．

解析対象のすべてのアプリに対して手動での詳細な動的解析を行うことはコストが高く，また，自動化された動的解析によって，アプリの持つ機能を網羅的に検証することは難しいと考えられる．これらをふまえ，今回の実験では，自動，および手動での動的解析の両方を用いることで，段階的な検証を行った．

- 自動で行われた動的解析の手順

1. adb [8]（Android Debug Bridge）を用いて解析対象アプリのインストール・起動．
2. 60 秒間待機した後に画面のスクリーンショットを撮影．
3. （手法 3 の場合のみ）adb を用いて android.intent.action.BOOT_COMPLETED のインテントを発行し，検証用コードを起動．
4. 外部サーバに対して，通信が行われたか判断．
5. 対象アプリのアンインストール．

まず，上記の解析手順を自動化することにより，すべてのアプリに対して，アプリが起動したかどうか，また，挿入された検証用コードが動作したかどうか，さらに，リパッケージ後にアプリの起動直後の画面の表示に変化が生

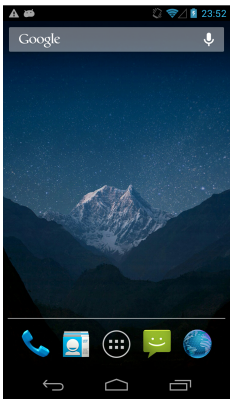

図 3 デフォルトのホーム画面が表示されている場合
Fig. 3 Default home screen.

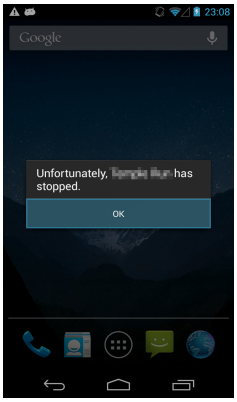

図 4 エラーメッセージが表示されている場合
Fig. 4 Error dialog on the screen.

じたかどうか，の 3 点について検証した．今回の実験における，アプリが正常に起動したかどうかの定義として，画面のスクリーンショットを用いた．具体的には，アプリの起動後の画面のスクリーンショットを目視で確認し，**図 3**，**図 4** のようにデフォルトのホーム画面や，エラーメッセージのダイアログが表示されていた場合を，アプリの起動の失敗，と定義した．動的解析中に，外部に用意したサーバに対して通信が発生したものについては，挿入した検証用コードが正常に動作したと判断した．さらに，リパッケージ前後の両アプリのスクリーンショットの比較を行い，アプリの起動直後の画面表示に変化が生じたかどうかを確認した．2 枚のスクリーンショットの画像の画素ごとの比較を行い，画素が 90%以上一致していた場合は，アプリ起動直後の動作が変化しなかったと判断した．ここで画素の一致率に対して閾値を設定しているのは，日時や広告表示などといった，リパッケージの有無にかかわらず，アプリの起動のタイミングによって画面上で変化する部分を無視するためである．画素の一致率が 90%以下であった場合には，目視での確認を行いながらリパッケージ前後の両アプリを起動し，本質的な画面表示が変化していないかどうかを主観的に判断した．

次に，起動直後の画面の表示以外で，アプリの持つ機能に変化が生じていないかどうかを確認するために，手動での動的解析を行った．今回は実験の簡略化のために，自動的な動的解析の結果において，アプリが正常に起動し，かつ検証用コードが動作し，さらにリパッケージ後に起動直後の画面の表示が変化しなかったものの中から，アプリ収集時に確認したユーザによるインストール数が 5,000 万を超えていた 33 種類のアプリを手動での動的解析の対象とした．手順としては，まず，リパッケージ前のアプリを適切な操作を加えながら動的解析し，それぞれのアプリの持つ各種機能の動作を確認した．この際，アプリが持つすべての機能を網羅的にチェックすることは難しいため，通常使用の範囲内で利用される可能性が高いと思われる機能を優先的にチェックした．これらの機能を検査項目として，リパッケージ後のアプリに対しても同様に動的解析を行い，各種機能の動作に変化が生じていないかどうかを主観的に判断した．

表 3 正規アプリの動的解析結果
Table 3 Result of automated dynamic analysis (Before repackaging).

	アプリ数*1
収集した正規アプリ	1612 (100%)
端末へのインストールに成功	1549 (96.1%)
アプリの起動に成功	1520 (94.3%)

*1 括弧内は収集した正規アプリに対する割合

4.3 実験結果

4.3.1 収集した正規アプリの動的解析

収集した正規アプリのリパッケージを行う前の動的解析の結果を**表 3** に示す．収集した 1,612 種類の正規アプリのうち，最終的に 1,520 種類のアプリが解析環境の端末上で起動に成功した．起動に失敗した 92 種類のアプリに関しては，リパッケージの前後のアプリの挙動を比較することができないため，自動リパッケージの対象から除外した．

4.3.2 自動リパッケージ

自動リパッケージにおける，ディスアセンブル，検証用コードの挿入，再ビルド，署名付与の各処理に成功したアプリの内訳を**表 4** に示す．今回自動リパッケージ対象としたアプリの中には，そもそも apktool によるディスアセンブルに失敗するものが 4 種類存在した．また，検証用コードの挿入時には，手法 1 においてのみ，111 種類のアプリに対して処理が失敗した．これらに関しては，onCreate メソッドの先頭部分を特定する際に用いた文字列マッチングの結果，コード挿入場所を見つけることができなかった場合であった．さらに，検証用コードの挿入後，apktool によって再ビルドを行う際にエラーが発生し，処理に失敗する検体が，手法 1 においては 100 種類，手法 2 では 137 種類，手法 3 では 56 種類存在した．署名の付与については，

表 4 自動リパッケージにおける各処理に成功したアプリの内訳
Table 4 Result of each process in automated repackaging.

	アプリ数*2		
	手法 1	手法 2	手法 3
自動リパッケージ対象とした正規アプリ	1520 (100%)		
apktool によるディスアセンブルに成功	1516 (99.7%)		
検証用コードの挿入に成功	1405 (92.4%)	1516 (99.7%)	
再ビルド, jarsigner による署名付与に成功	1305 (85.9%)	1379 (90.7%)	1460 (96.1%)

*2 括弧内は自動リパッケージ対象とした正規アプリに対する割合

表 5 リパッケージ済み正規アプリの自動での動的解析結果
Table 5 Result of automated dynamic analysis (After repackaging).

		アプリ数*3		
		手法 1	手法 2	手法 3
自動リパッケージ対象とした正規アプリ		1520 (100%)		
動的解析対象としたリパッケージ済みアプリ		1305 (85.9%)	1379 (90.7%)	1460 (96.1%)
端末へのインストールに成功		1286 (84.6%)	1359 (89.4%)	1440 (94.7%)
アプリの起動に成功		1194 (78.6%)	1315 (86.5%)	1411 (92.8%)
外部サーバへの通信が発生		1206 (79.3%)	1351 (88.9%)	1415 (93.1%)
アプリの起動時の画面表示に変化が生じたかどうか	画面に変化無し	1169 (76.9%)	1290 (84.9%)	1381 (90.9%)
	画面に変化有り	136 (8.9%)	89 (5.9%)	79 (5.2%)
外部サーバへ通信が発生し，尚且つ画面表示に変化が生じなかったもの		**1162 (76.4%)**	**1286 (84.6%)**	**1368 (90.0%)**

*3 括弧内は自動リパッケージ対象とした正規アプリに対する割合

すべての手法の全検体に対して，問題なく処理が成功した．

各手法における結果を比較してみると，すべての処理に成功したアプリの割合は，手法 1 では 85.9%であったのに対して，手法 2 では約 90.7%，手法 3 では 96.1%と，手法 1 より高い成功率となった．

各手法において，すべての処理に成功したアプリに対して動的解析を行い，挙動を確認した．

4.3.3 リパッケージ済み正規アプリの自動での動的解析

リパッケージ済み正規アプリの自動的に行われた動的解析の結果を**表 5** に示す．自動リパッケージのすべての処理に成功したアプリの中で，手法 1 では 1,289 種類，手法 2 では 1,359 種類，手法 3 では 1,440 種類のものが，解析環境の端末へのインストールには成功したが，各手法において，19〜20 種類の検体は “INSTALL_PARSE_FAILED_MANIFEST_MALFORMED” のエラーが原因でインストールに失敗した．これは，インストール対象のアプリの AndroidManifest.xml の形式が不正である場合に発生するエラーであり，リパッケージの処理の際に，AndroidManifest.xml の改変に失敗していることが予想される．また，最終的に外部サーバへの通信が確認できたものが，手法 1 においては 1,206 種類，手法 2 においては，1,351 種類，手法 3 においては 1,415 種類存在した．これらのアプリに関しては，挿入した検証用コードが正常に動作していると考えられる．さらに，検証用コードは正常に動作したが，アプリの起動には失敗しているパターンが手法 1 では 24 種類，手法 2 では 42 種類，手法 3 では 22 種類のアプリにおいて確認された．これは，検証用コードが実行された後に，正規アプリコード部分でエラーが発生し，アプリが強制終了している場合であると考えられる．

リパッケージ前後の，各アプリの挙動に関しては，各手法において 7 割以上が，自動リパッケージを行っても，起動後の画面表示に変化がないことが分かる．最終的に，外部サーバへ通信が発生し，なおかつリパッケージ後に起動後の画面表示の機能に変化がなかったものが，手法 1 において 76.4%，手法 2 においては 84.6%，手法 3 においては 90.0%存在した．

いずれの手法を用いても，挿入した検証用コードが動作し，なおかつ画面の表示に変化がなかったアプリが 1,078 種類存在した．今回は，それらの中から，ユーザによるインストール数が 5,000 万回以上であった，33 種類のアプリに対して，手動での動的解析を行った．

表 6 リパッケージ済みアプリの手動での動的解析結果
Table 6 Result of manual dynamic analysis (After repackaging).

	アプリ数*4
手動での動的解析対象としたアプリ	33 (100%)
すべての検査項目において，アプリの機能が保持されていたもの	29 (87.9%)
一部の検査項目において，アプリの機能に変化が生じていたもの	4 (12.1%)

*4 括弧内は手動での動的解析対象としたアプリに対する割合

4.3.4 リパッケージ済み正規アプリの手動での動的解析

手動での動的解析の結果を**表 6** に示す．手動での動的解析を行ったアプリのうち 87.7%のものは，すべての検査項目において，元の正規アプリの機能が保持されていた．一

方で，4 種類においては，一部の機能が保持されておらず，結果としてアプリを本来の目的において使用することができない状態となっていた．4 種類のアプリにおいて保持されなかった機能は，アップデート機能や，サービスへのログイン機能など，いずれも外部のサーバとの通信が発生する機能であった．

なお，手動での動的解析の際の検査項目，および検査結果について，アプリの機能が保持されなかった 4 種類，およびそれ以外から抜粋した 4 種類のものを末尾の付録に添付する．

5. 考察

5.1 正規アプリの自動リパッケージへの耐性

今回の検証では，自動リパッケージによって長期的に感染を続けるようなマルウェアを作成する攻撃者を想定している．攻撃者の視点から考えると，元の正規のアプリと同じ機能を提供することでユーザに長期的にアプリを使用させつつ，その裏で挿入した悪性コードを定常的に動作させ続けることを目的としていると予想される．これらをふまえると，下記の 2 つの条件のどちらかにあてはまるアプリが，今回想定したような目的で行われる自動リパッケージに対して耐性があるといえる．

(1) 挿入された悪性コードが動作しない場合
(2) 挿入された悪性コードは動作するが，元の正規アプリの機能が損なわれている場合

また，悪性コードが動作する期間の長さから考えると，悪性コードがまったく動作しない (1) の条件を満たすものが最も耐性が強く，(2) の条件を満たすアプリの中でも，正規アプリの機能が損なわれることにより，より早いタイミングで悪性コードが動作しなくなるものの方が耐性が強いといえる．加えて，上記の (1)，(2) の条件すべてにあてはまらないアプリについては，今回想定する攻撃に対しては，まったく耐性がないといえる．

前章の実験結果を見てみると，用いた手法によって結果に差があるものの，全体の 7 割から 9 割のアプリにおいて，挿入した検証用コードが正常に動作し，なおかつ起動後の画面の表示に変化が生じていないことが分かる．これらのアプリについては，上記の (1) にはあてはまらず，また，少なくとも起動直後の画面表示に関わる機能は損なわれていないため，そもそもリパッケージ処理に失敗したアプリや，起動に失敗するようになったアプリと比較して，自動リパッケージへの耐性が低いといえる．さらに，それらの中から抽出した 33 種類のアプリのうち，29 種類については，手動での動的解析の結果から，元のアプリの機能が保持されていることが分かる．これらの 29 種類については，今回の検証の範囲内では，上記の (1)，(2) のどちらの条件にもあてはまらないため，自動リパッケージに対する耐性がないといえる．以上より，現状の Android アプリの多くは自動リパッケージへの耐性が不十分であり，リパッケージ元の正規アプリと同様に動作し，その裏で悪意のあるコードを実行するマルウェアが大量に作成される可能性が十分に考えられる．

5.2 アプリの動的解析手法について

今回は実験の簡略化のために，アプリの動的解析の際に，アプリが正常に起動したかどうかの判断や，リパッケージによって元の正規アプリから起動直後の動作が変化しないかどうかを，スクリーンショットを用いた機械的な方法で判断した．一方，起動後に画面の表示に変化は生じるが，画面の表示以外の内部処理でエラーが発生し実際には正常にアプリが起動していない場合が考えられる．さらに，ユーザによるタッチなどのイベントによってアプリの内部処理に分岐が生じる場合や，後述するリパッケージ対策がアプリの起動直後以外の場所に施されている場合も考えられる．上記を考慮することで，より細かい粒度での自動リパッケージへの耐性を検証することが可能であるが，一方で実験のコストから考えると，自動化可能でなおかつより詳細にアプリの挙動を把握可能である動的解析手法が必須であると考えられる．具体的には，デバッギング技術などを用いてより詳細なログを取得する手法や，動的解析時のコードカバレッジを向上させるような手法の組合せなどが考えられる．

5.3 自動リパッケージに失敗したアプリについて

リパッケージにおける各処理や，コード挿入後の動的解析時において何らかのエラーが発生した検体が存在した．**表 7**，**表 8** にそれぞれのエラーが発生したパターンと検体数についてまとめる．

a. アプリのリパッケージ処理（ディスアセンブル〜署名付与）で発生したエラーについて

(a-1) apktool によるディスアセンブルに失敗したケースについて調査した結果，apktool のバグに起因していることが判明した．これらについては，今後の apktool のアップデートによって処理に成功するようになる可能性がある．

(a-2) 手法 1 においてコード挿入位置が特定できなかったケースについては，マッチングに用いた文字列が不適切であった場合，つまり特定の実装パターンにおいては対象とした文字列が smali コード内に現れない場合があったためであると考えられる．また，今回用いた文字列検索を，smali コードを厳密に解釈して挿入位置を決定するアルゴリズムに変更することで，より多くの検体に対して処理が成功することが予想されるが，一方で，そのような高度な処理を自動化するのはコストが高いため，成功率とコストのトレードオフになることが予想される．

表 7 リパッケージ処理中に発生したエラーの内訳
Table 7 Details of error happened while repackaging.

	アプリ数*5		
	手法 1	手法 2	手法 3
リパッケージ処理中にエラーが発生した全アプリ	215 (100%)	141 (100%)	60 (100%)
(a-1) apktool によるディスアセンブルに失敗	4 (1.9%)	4 (2.8%)	4 (6.7%)
(a-2) 検証用コードの挿入に失敗	111 (51.6%)	0 (0%)	0 (0%)
(a-3) apktool による再ビルドに失敗	100 (46.5%)	137 (97.2%)	56 (93.3%)

*5 括弧内はリパッケージ処理中にエラーが発生した全アプリに対する割合

表 8 動的解析時に発生したエラーの内訳
Table 8 Details of error happened while dynamic analysis.

	アプリ数*6		
	手法 1	手法 2	手法 3
動的解析時にエラーが発生した全アプリ	143 (100%)	93 (100%)	92 (100%)
(b-1) 端末へのインストールに失敗	19 (13.3%)	20 (21.5%)	20 (21.7%)
(b-2) アプリの起動に失敗	93 (65.0%)	44 (47.3%)	29 (31.5%)
(b-3) 検証用コードが動作せず【b-1 以外】	80 (55.9%)	8 (8.6%)	25 (27.2%)
(b-4) アプリの挙動が変化【b-1,b-2 以外】	24 (16.8%)	26 (28.0%)	30 (32.6%)

*6 括弧内は動的解析中にエラーが発生した全アプリに対する割合

(a-3) apktool による検証用コード挿入後の再ビルド時にエラーが発生したケースについては，様々な原因が考えられるが，多くの場合で，挿入されたコードが原因で，smali コードのフォーマットに不整合が生じたために，dex 形式へと再ビルドすることができなかったためであると考えられる．

b. 動的解析時に発生したエラーについて

(b-1) リパッケージ後に端末へのインストールに失敗するようになった検体について調査した結果，リパッケージ処理において AndroidManifest.xml の改変に失敗しており，結果として AndroidManifest.xml が不正な状態で再ビルドが行われていたことが判明した．これらのエラーはリパッケージを行うスクリプトの改良によって改善されると考えられる．

(b-2) リパッケージ後にアプリの起動に失敗するようになったケースについては，アプリの起動時にエラーが発生し，強制終了している場合と，エラーは発生していないが，アプリを起動させてもデフォルトのホーム画面から変化が生じない 2 つのパターンが存在した．

(b-3) 挿入した検証用コードが動作しなかったケースについて，検証用コードが動作せず，なおかつアプリの起動に失敗している場合が，手法 1 では 69 種類，手法 2 では 2 種類，手法 3 では，7 種類のアプリにおいて確認された．これらについては，処理が検証用コードに到達する前に，エラーが発生し，アプリが強制終了したパターンが原因として考えられる．各手法において，上記のようなパターンであったアプリ数に差がみられるが，これは，それぞれの手法における検証用コードの呼び出され方の違いに起因していることが予想される．

(b-4) アプリのインストール，および起動には成功するが，リパッケージ前と比較してアプリの動作に変化が生じたケースについては，起動時に外部に用意されたサーバと通信するタイプのアプリにおいて，通信が正常に行われなくなるパターンや，リパッケージ後にのみポップアップによる通知などが現れるパターンなどが存在した．

上記のエラーのうち，(a-1)，(b-1) については，リパッケージ処理の不備が主なエラーの原因であると考えられるが，一方で，(a-2)，(a-3)，(b-2)〜(b-4) については，アプリになんらかのリパッケージ対策がなされていたために処理に失敗した可能性も考えられる．特に (b-4) については，5 種類のアプリにおいて明確にリパッケージ対策がなされていることを確認した（5.6 節を参照）．

5.4 検証用コードの挿入方法による自動リパッケージの成功率の差異

今回の実験では，検証用コードの挿入時に 3 つの手法を用いたが，各手法ごとに自動リパッケージの成功率に差がみられた．**表 9** は各手法において自動リパッケージの際に行った処理の内容を比較したものである．最も成功率が低かった手法 1 は，元の正規アプリに含まれていた実行コードに対する改変を行っており，これが前節における (a-2)，(a-3)，(b-2) のようなエラーの発生数を増加させる要因になっていることが予想される．また，手法 2 は全手法の中で検証用コードが呼び出されるタイミングが最も早いため，(b-3) のようなエラーが発生しにくいといえる．さらに，手法 3 では，パッケージの追加，コンポーネント・パーミッション定義の追加のみを行っており，他の手法と比較して，元の正規アプリに含まれていたコード・リソースへの改変が少ないため，全体として，処理に失敗する検体数が少なかったと考えられる．

表 9 各手法において自動リパッケージの際に行った処理の比較
Table 9 Comparison of procedures in each repackaging method.

処理内容		手法 1	手法 2	手法 3
元の正規アプリに含まれていた実行コード(smali ファイル)の改変		✔	-	-
AndroidManifest.xml の改変	既存のコンポーネント定義の改変	-	✔	-
	コンポーネント・パーミッションの追加	✔	✔	✔
新規パッケージの追加		✔	✔	✔

5.5 自動リパッケージの手法と挿入されたコードのエントリポイントについて

今回の実験では，過去に流行した 3 種類のリパッケージマルウェアの中で実際に用いられていたリパッケージ手法を模擬したが，それぞれの手法は，挿入されたコードの呼び出され方が異なっていた．自動化が可能なリパッケージの手法は，挿入されるコードが何をトリガとして実行されるか，という観点からいくつかのパターンに分類可能であると考えられる．

まず，ユーザによるアプリの起動をトリガとして挿入されたコードが実行される手法を考える．通常，アプリが起動する際には，AndroidManifest.xml 内で android.action.MAIN のインテントフィルタが指定されたアクティビティが呼び出される．アクティビティ起動後の処理は各々のアプリの実装に依存してしまうため，自動化が可能な方法としては，(a) 起動直後に呼び出される正規アプリのアクティビティを機械的に改ざんし，悪性コードのエントリポイントを挿入する方法，(b) android.action.MAIN の受け取り先自体を変更し，任意の処理を行った後に，明示的インテントなどを用いて元々呼び出されるはずであった正規アプリのアクティビティを呼び出す方法，の 2 種類が考えられる．本実験における手法 1，手法 2 はそれぞれ (a)，(b) に分類される．

アプリの起動以外でコード実行のトリガになりうるものとして，ブロードキャストインテントが考えられる．Android では，たとえば端末の再起動や，SMS の受信など，端末上で発生したイベントに応じてアプリが任意のコードを実行するための仕組みとして，ブロードキャストインテントが用いられる．ブロードキャストインテントを利用した自動化可能なリパッケージの手法としては，(c) AndroidManifest.xml 内に新たにブロードキャストインテントを受け取るための BroadcastReceiver を定義し，そこをエントリポイントとして任意の処理を実行する方法が考えられる．本実験における手法 3 が上記 (c) に分類される．

リパッケージの手法が自動化可能であるかどうかを考える際には，挿入されるコードを呼び出すためのエントリポイントにあたる部分を機械的に挿入することができるか，という問題に帰着することが予想される．よって今回利用したアプリの起動や，ブロードキャストインテント以外のエントリポイントであっても機械的な処理で挿入可能であれば，自動リパッケージに利用される可能性があるといえる．

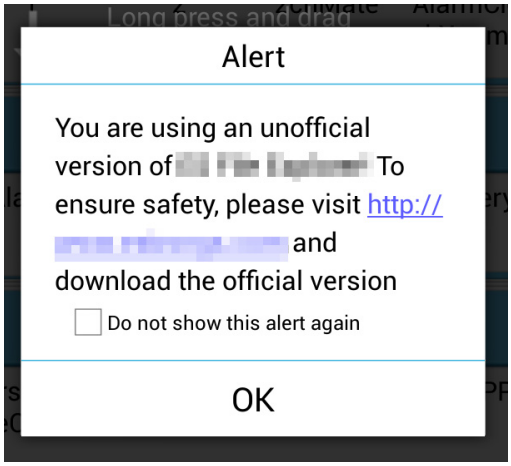

図 5 リパッケージ済みアプリの動的解析時に表示されたダイアログ
Fig. 5 Dialog that only appears after repackaging.

5.6 リパッケージ対策が施されていたアプリについて

今回の評価対象とした正規アプリの中で，リパッケージへの対策がなされていたと思われるものが見受けられたため，以下にまとめる．

リパッケージ後のアプリを動的解析した際に，**図 5** のようなダイアログが表示されるものが存在した．これは，アプリが自身の改ざんを検知して，ユーザに対して注意喚起を行っていると考えられる．今回の実験中に，5 種類のアプリが，同様のポップアップ表示によってユーザに対し注意喚起を行っているのを確認した．このような改ざん検知機能によって，ユーザはマルウェアの存在に気付きやすくなり，端末内へ長期間潜伏するようなタイプのマルウェアの被害を軽減することができると考えられる．改ざんを検知するための具体的な方法としては，アプリに施された署名情報のチェックや，実行コードやリソースファイルのハッシュ値のチェックなどが考えられる．

5.7 耐タンパ化によるリパッケージ対策

自動リパッケージへの対策としては，前節で例示した改ざん検知などの一般的な耐タンパ技術が有効であると考えられる．Android に適用可能な耐タンパ技術として，文献 [9] で提案されている手法があげられる．文献 [9] 内で

は，Android NDK [10] を用いて Android 上で自己書き換え型の耐タンパ技術を実現している．Android NDK において，実行コードは Java ではなく，より解析が困難であるネイティブコードで記述されるため，前述したようなアプリの改ざん検知機能を Android NDK を用いて実装することで，攻撃者による自動リパッケージを，より困難にすることが可能である．

5.8 リパッケージ対策の今後の展望

上記のような対策が考えられる一方で，技術レベルの高い攻撃者が手動でリパッケージを行った場合には，Android におけるリバースエンジニアリングの容易さなどから，それらを完全に防ぐことは難しいと考えられる．しかし，一般的に，個々のアプリに対応した高度なリパッケージの処理を自動化することは困難であるため，今回検証を行ったような自動リパッケージを防ぐための対策を行うだけでも十分に被害を軽減できると考えられる．多くの正規アプリにおいて，自動リパッケージの対策がなされていない現状をふまえると，たとえば，自動的にアプリに対して耐タンパ化を施してくれるようなツールや，耐タンパ性を高める開発方法のマニュアル化などといった，セキュリティに関する知識が十分でないアプリ開発者でも，「自動リパッケージを防ぐための機能」をアプリに容易に付与できる対策技術が必要であるといえる．

一方で，上記のような対策技術が一般化すると，これを自動的に回避する攻撃が現れ，対策が陳腐化し，いわゆる「いたちごっこ」の状態に陥ってしまうことが予想される．しかし，攻撃側は自動リパッケージを行う対象のアプリをブラックボックスとして扱わなければならないのに対して，対策を行う側（開発側）は，その内容やソースコードを所持しており，守る側に有利である．今後は，そういった優位性を活かすことで，陳腐化しにくい自動リパッケージ対策を検討していく必要がある．

5.9 正規アプリが要求するパーミッション

今回評価対象とした正規アプリのほとんどが，ネットワーク接続のためのパーミッションを要求していた．また，GET_ACCOUNTS や READ_PHONE_STATE などといった，ユーザの個人情報にアクセスする際に必要とされるパーミッションをあわせて要求しているものも見受けられた．たとえば，電話帳アプリに対して連絡先情報を漏洩する悪性コードを挿入する場合や，地図アプリに対して位置情報を漏洩させる悪性コードを挿入する場合などのように，攻撃者が，悪性コードに実装した機能に応じてリパッケージ対象の正規アプリを選ぶことで，新たにパーミッションを加えることなくリパッケージが行われる可能性がある．これらのリパッケージマルウェアは，ユーザがパーミッションの確認を行っても不審に見えないため，より危険性が高いといえる．

5.10 Android マルウェアの流通・流行に関して

攻撃者が自動リパッケージを行って大量のリパッケージマルウェアを作成した後に，どのような経路でユーザの端末へ感染させるかについても今後調査が必要である．攻撃者は多くの場合，アプリ掲載時の審査が不十分なサードパーティマーケットにおいてリパッケージマルウェアを配布していると考えられる．先行研究 [2] では，マーケットプレイス上でリパッケージマルウェアを検出する手法が提案されており，今後，リパッケージマルウェアに対する対策を考えるうえで，前述した耐タンパ化のような，各アプリの作成者が主体となって行う対策に加えて，マーケットを運営しているキャリアやベンダが主体となって行う対策も組み合わせていくことが必要と考えられる．

6. まとめと今後の課題

本論文では，Android アプリケーションの自動リパッケージに対する耐性評価実験を行った．その結果，用いた手法により成功率に差がみられたものの，評価対象とした既存の正規アプリのうち，7〜9 割のものに対して，起動時の画面表示機能に影響を与えることなく検証用コードを挿入することが可能であることを示した．さらに，多くのユーザによって利用されている，人気のアプリにおいて，リパッケージを行ってもアプリの持つ基本的な機能に変化が生じない例を確認した．今回挿入した検証用コードを，悪性な動作を行うコードに置き換えることは容易であるため，現状の Android アプリケーションの多くは，攻撃者によって容易にリパッケージが可能であり，リパッケージ元の正規アプリと同様に動作し，その裏で悪性コードを実行することで，長期的に感染を続けるようなマルウェアが大量に作成される危険性があるといえる．

自動リパッケージへの対策としては，アプリの改ざんを検知する機能などが有効であると考えられる．一方，攻撃者が高度な技術を用いてリパッケージを行った際には，それらを完全に防ぐことは難しいが，自動的なリパッケージを防ぐ対策は攻撃者のコストを増加させる点で被害を軽減する効果が期待できる．

今後は，陳腐化しにくい自動リパッケージ対策手法の検討や，マーケットプレイスごとに行う対策など，より広い視点からのリパッケージ対策についても検討していきたいと考える．

謝辞 本研究の一部は，総務省情報通信分野における研究開発委託/国際連携によるサイバー攻撃の予知技術の研究開発/サイバー攻撃情報とマルウェア実体の突合分析技術/類似判定に関する研究開発により行われた．また，本研究の一部は，文部科学省国立大学改革強化推進事業の支援を受けて行われた．

参考文献

[1] MDK：中国で最大のモバイルボットネット | Symantec Connect コミュニティ，入手先 ⟨http://www.symantec.com/connect/ja/blogs/mdk⟩（参照 2013-12-04）.

[2] Zhou, W., Zhou, Y., Jiang, X. and Ning, P.: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces, *2nd ACM Conference on Data and Application Security and Privacy* (*CODASPY 2012*) (2012).

[3] Huang, H., Zhu, S., Liu, P. and Wu, D.: A Framework for Evaluating Mobile App Repackaging Detection Algorithms, *6th International Conference on Trust & Trustworthy Computing* (*Trust 2013*) (2013).

[4] Intents and Intent Filters | Android Developers, available from ⟨http://developer.android.com/guide/components/intents-filters.html⟩ (accessed 2013-12-09).

[5] android-apktool – A tool for reverse engineering Android apk files – Google Project Hosting, available from ⟨http://code.google.com/p/android-apktool/⟩ (accessed 2013-12-09).

[6] smali – An assembler/disassembler for Android's dex format – Google Project Hosting, available from ⟨http://code.google.com/p/smali/⟩ (accessed 2013-12-09).

[7] jarsigner – JAR Signing and Verification Tool, available from ⟨http://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html⟩ (accessed 2013-12-09).

[8] Android Debug Bridge | Android Developers, available from ⟨http://developer.android.com/tools/help/adb.html⟩ (accessed 2013-12-09).

[9] 吉田直樹, 吉岡克成, 松本　勉：自己書換え型耐タンパー技術のスマートフォン環境における検証, 信学技報, Vol.113, No.135, ISEC2013-13, pp.19–26 (2013).

[10] Android NDK | Android Developers, available from ⟨http://developer.android.com/tools/sdk/ndk/index.html⟩ (accessed 2013-12-09).

付　　録

表 A·1 手動での動的解析の際の検査項目，および検査結果（アプリの機能が保持された場合，抜粋）
Table A·1 Detailed result of manual dynamic analysis (Original function is remained, Excerpt).

アプリの種類（インストール数）	検査項目としたアプリの機能	リパッケージ後の動作（◯：変化無し，×：変化有り）			備考
		手法 1	手法 2	手法 3	
ゲームアプリ A（5000 万〜1 億）	ランチャーからアプリを起動	◯	◯	◯	
	基本的な UI の操作	◯	◯	◯	
	オフラインプレイ	◯	◯	◯	
	オンラインプレイ	◯	◯	◯	
ゲームアプリ B（1 億〜5 億）	ランチャーからアプリを起動	◯	◯	◯	
	基本的な UI の操作	◯	◯	◯	
	ゲームプレイ	◯	◯	◯	
	SNS 連携	◯	◯	◯	
Web ブラウザアプリ（5000 万〜1 億）	ランチャーからアプリを起動	◯	◯	◯	
	基本的な UI の操作	◯	◯	◯	
	Web ページ閲覧	◯	◯	◯	
SNS アプリ（1 億〜5 億）	ランチャーからアプリを起動	◯	◯	◯	
	基本的な UI の操作	◯	◯	◯	
	ログイン	◯	◯	◯	
	投稿	◯	◯	◯	

表 A·2 手動での動的解析の際の検査項目，および検査結果（アプリの機能に変化が生じた場合，全 4 種）
Table A·2 Detailed result of manual dynamic analysis (Original function is not remained, 4 apps).

アプリの種類（インストール数）	検査項目としたアプリの機能	リパッケージ後の動作（◯：変化無し，×：変化有り）			備考
		手法 1	手法 2	手法 3	
Android 用 AV ソフト（1 億〜5 億）	ランチャーからアプリを起動	◯	◯	◯	
	基本的な UI の操作	◯	◯	◯	
	AV スキャン	◯	◯	◯	
	ウイルス定義更新	×	×	×	ウイルス定義の更新に失敗
天気情報アプリ（5000 万〜1 億）	ランチャーからアプリを起動	◯	◯	◯	
	基本的な UI の操作	◯	◯	◯	
	天気情報表示	◯	◯	◯	
	地図情報の表示	×	×	×	地図の表示に失敗
フォトエディター（1 億〜5 億）	ランチャーからアプリを起動	◯	◯	◯	
	基本的な UI の操作	◯	◯	◯	
	画像編集機能	◯	◯	◯	
	ソーシャル機能：ログイン	×	×	×	サーバとの通信に失敗
無料通話・メッセンジャーアプリ（1 億〜5 億）	ランチャーからアプリを起動	×	×	×	起動時にアプリのアップデートに失敗

金井 文宏

2015 年 3 月横浜国立大学大学院環境情報学府情報メディア環境学専攻博士課程前期修了，修士（情報学）．同年 4 月より日本電信電話株式会社, NTT セキュアプラットフォーム研究所社員．同社サイバーセキュリティプロジェクト，セキュアネットワーク構成グループ所属．Android マルウェアの解析・検知やネットワーク攻撃の分析等，モバイル・ネットワークセキュリティの研究に従事．

庄田 祐樹

2013 年 3 月横浜国立大学工学部電子情報工学科卒業．同年 4 月同大学大学院博士課程前期進学．Android マルウェアの解析等のモバイルセキュリティの研究に従事．2015 年 3 月横浜国立大学大学院環境情報学府情報メディア環境学専攻博士課程前期修了，修士（情報学）．

橋田 啓佑

2014 年 3 月横浜国立大学工学部電子情報工学科卒業．同年 4 月同大学大学院環境情報学府情報メディア環境学専攻博士課程前期進学，在籍中．Android マルウェアの解析環境に関する研究等のモバイルネットワークセキュリティの研究に従事．

吉岡 克成 （正会員）

2005 年 3 月横浜国立大学大学院環境情報学府情報メディア環境学専攻博士課程後期修了，博士（工学）．同年 4 月独立行政法人情報通信研究機構研究員．2007 年 12 月より横浜国立大学学際プロジェクト研究センター特任教員（助教）．2011 年 4 月より横浜国立大学大学院環境情報研究院准教授．マルウェア解析やネットワーク攻撃観測・検知等のネットワークセキュリティの研究に従事．2009 年文部科学大臣表彰・科学技術賞（研究部門）受賞．

松本 勉 （正会員）

1986 年 3 月東京大学大学院工学系研究科電子工学専攻博士課程修了，工学博士．同年 4 月横浜国立大学講師．2001 年 4 月より同大学大学院環境情報研究院教授．2014 年 12 月より同大学先端科学高等研究院主任研究者を兼務．ネットワーク・ソフトウェア・ハードウェアセキュリティ，暗号，耐タンパ技術，生体認証，人工物メトリクス等の「情報・物理セキュリティ」の研究教育に 1981 年より従事．1982 年にオープンな学術的暗号研究を目指した「明るい暗号研究会」を 4 名で創設．2005～2010 年国際暗号学会 IACR 理事．1994 年第 32 回電子情報通信学会業績賞，2006 年第 5 回ドコモ・モバイル・サイエンス賞，2008 年第 4 回情報セキュリティ文化賞，2010 年文部科学大臣表彰・科学技術賞（研究部門）受賞．