モバイルアプリケーションにおける ファイルヘッダー情報に着目した不正プログラムの検知

草間 好輝[†1,†2,*] 武田 圭史[†1] 中村 修[†1] 小林 和真[†1,†3] 李 明宰[†2]

概要：スマートフォン普及に伴い，モバイルマルウェアによる被害も増加している． 最近の傾向として，独自にマルウェアを作成するより，既存のアプリケーションを改変するタイプのマルウェアが増加している．これらのマルウェアは既存のアプリケーションと特徴が似ているため，検知することが難しい傾向にある． 本研究では Android Platform で用いられる DEX ファイルと Manifest ファイルの二つに対して機械学習を用いることで評価し，不正プログラムの検知を行った．DEX ファイルについては，標準的なコンパイラと，それ以外のコンパイラで作成されたものの違いに着目し， Manifest ファイルに関しては，Android に対して要求されているパーミッション情報を複数の学習アルゴリスムの入力パラメーターとして適応した． これらの情報を複数の学習アルゴリズムへ適応したとき，パーミッション情報のみの機械学習を適応した検知方法と比べて 5%ほど精度，F 値が向上した． また誤検知率に関しても大幅に軽減することが確認できた．

キーワード：マルウェア，機械学習，アンドロイド，ヘッダー情報，DEX

Malware Detection Based on the File header Information In Mobile Application

Yoshiki Kusama[†1,†2,*] Keiji Takeda[†1] Osamu Nakamura[†1] Kazuma Kobayashi[†1,†3] Myeongjae Lee [†2]

Abstract: The damage caused by mobile malware also has been increasing With the spread of Smartphones . As a recent trend, the type of malware that modifies existing applications is increasing rather than creating original malware. These malware tend to be difficult to detect because they have similar characteristics to existing Applications. In this study, we evaluated the DEX file and Manifest file used in the Android Platform by using machine learning to detect malicious programs. The DEX file focuses on the differences between the standard compiler and those created by other compilers, and the Manifest file adapts the permission information required for Android as input parameters for multiple learning algorithms. When this information was applied to multiple learning algorithms, the Accuracy and F-meature improved by about 5% compared to the detection method that applied machine learning using only permission information. It was also confirmed that the false positive rate was significantly reduced.

Keywords: Malware, Machine Learning, Android, Header Infromation, DEX

1. はじめに

スマートフォンは所有率が 82.2%[1]と，今では生活に欠かすことが出来ない重要なアイテムとなっている． その中でも Android Operating System はスマートフォンで最も用いられている Mobile Operating System であり，約 85.9%の市場シェアを占めている[2]. 一方，プラットフォーム上の Application におけるセキュリティの脅威は年々増加の意図を辿っており，悪意のあるアプリケーション４による被害が後を立たない．これは Android では Third-Party Store などからの Application のインストールを許可していることが，増加の原因の一つだと考えられている．

まず Android 端末上でアプリケーションとしてインストールする APK と，Repackage に関して説明する．

1.1 APK(Android Application PacKage)

APK(Android Application PacKage)は Android 上で動作するファイルフォーマットであり，JAR ファイルをベースとした ZIP 形式のアーカイブファイルである．

このアーカイブファイルにはアンドロイド上で動作に必要な様々なファイルが納められており，これらのファイルを Android 上でインストールできる形式にパッケージにしている． 図 1 に APK の内部構造を示す. APK は大きく分けて，*AndroidManifest.xml*，*META_INF*，*lib/*，*Asset/*，*Classess.dex*，*res/*，*resources.arsc* で構成されている． これらのファイルにはそれぞれ役割が存在する．機能を以下に示す.

† 1 慶應義塾大学大学院 Keio University
† 2 LINE 株式会社 LINE Corporation
† 3 奈良先端科学技術大学院大学 Nara Institute of Science Technology
* ykusama@keio.jp

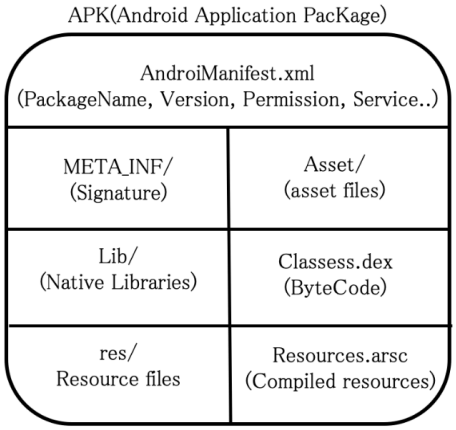

図 1. APK 内部構造

META-INT には基本的に署名ファイルが格納されている. 例えば APK のチェックサム(SHA-1)や, アクセス制限のリスト, *res* や*assets* に格納されているファイルのチェックサムなどを記録したファイルがある.

res には基本的にリソースファイルが含まれている. Android へインストールした時に表示される, アンコンのなどがこれに含まれまれる.

Assets には *res* フォルダには入らない, txt ファイルや, ZIP ファイル, バイナリファイルなどがこのフォルダへ格納される. *Lib* にはコンパイルされたネイティブコードが格納される. ARM や x86 など, 様々なアーキテクチャー用にコンパイルされたコード(**.so*)が格納される.

Classess.dex は Dalvik Runtime もしくは Android Runtime 上で動作するようなバイナリファイルである. Android プラットフォーム上ではレジスタベースの仮想環境(Dalvik RT, Androi RT)が動作している. 我々が JAVA で書いた高級言語は, 通常 Android Studio などで標準実装されている dx というコンパイラで, Runtime 環境で動作するような形式(**.dex*)にコンパイルされる. 図 2 にコンパイルプロセスを示す.

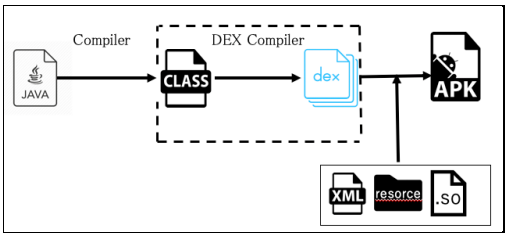

図 2. Compiler Process

図 2 のように, コンパイルされた *dex* は, AAPT(Android Asset Packaging Tool)によって, Resource ファイルなどとパッケージングされる.

AndroidManifest.xml はアプリケーションのタイトル, バージョン, アクセス制限などが含まれる. アプリケーションにおいては必須のファイルであり, ここにパーミッション情報などが格納される. Android は Linux Kernel をベースとしたシステムのため, Android システム自体がシステムとアプリケーションを保護するための一連のセキュリティメカニズムが存在する. 例えば, オペレーティングシステムごとに, プロセスの分離や, メモリ管理などが導入されており, 各アプリケーションは個別のシステム ID で動作する. そのため, 特定のプロセスを実行する具体的な操作に対して制限を設けるパーミッションなど, きめ細かいセキュリティ機能がある. Android アプリは, それぞれプロセスがサンドボックス内で動作するため, リソースやデータを明示的に共有しなければならない. 具体的には,ベーシックサンドボックスでは提供されない追加機能が必要な場合は, パーミッションを宣言する. アプリで必要なパーミッションを静的に宣言すると, Android システムがユーザーに同意を求める. この宣言が, *AndroidManifest.xml* 上に格納されており, パーミッションはおよそ 150 種類ある.

1.2 Repackage について

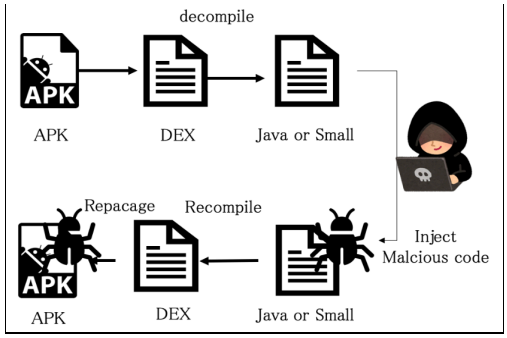

図 3. Repacage Process

図 3 のように攻撃者は通常, リバースエンジニアリングツールを用いて, APK ファイルのデコンパイルを行い, *dex* を java や small コードに直す. そして悪意のあるコードを挿入し, リコンパイル, リッパッケージを行う. ツールは *dex* を small ファイルに, デコンパイル, リコンパイルを行うことが出来る APKTOOL[3]や, *jar* を *java* に変換する dex2jar[4]などがある. これらのツールを使うことで, 通常の APK に悪意のあるコードを挿入することが可能となり, マルウェアを生成が可能となる. (以後：セカンドタイプマルウェアと呼ぶ)

2. 従来研究と問題

アンドロイドにおけるマルウェア解析技術には大きく分けて二つのアプローチが存在する. 一つ目に, マルウェアの挙動や動作によって分類する動的解析である. Wei 氏の論文[5]では, サンドボックスを用いて, アプリケーションの動的情報(ネットワーク情報, ファイル操作など)を抽出し, それらの情報を機械学習へ適応することにより, マルウェアの分類を行った. その結果, 通常のシステムより, 未知のマルウェアの検出確率が上昇することを確認できた.

Jiawei 氏の論文[6]では，システムコールの頻度情報に着目し，これらの情報を機械学習へ適応することによってマルウェアの分類を行なっている．その結果，51 個中 50 個のテストサンプルを正しく分類している．こういった動的解析は，マルウェアのコードを分析するといったリバースエンジニアリング等の高度な作業を必要としないといった利点が存在する．しかし一方で動的解析では，ボットなどのような決まった時刻にしか動作しないといったマルウェア等に関しては網羅的に挙動を抽出することは難しいといった欠点もある．

二つ目のアプローチとして，マルウェアのコードなどに着目し分析を行う静的分析がある．xlagyu 氏の論文[6]では，静的解析によって package 情報とパーミッション情報を抽出し，その情報を機械学習へ適応することによって，Android のマルウェアを分類している．その結果，パーミッション情報のみで検知を行った結果検知率は 80%ほどだが，Package 情報と組み合わせることで 2%ほど検知率が向上している． 最近では動的分析，席的分析の二つの情報に着目した，ハイブリットな検知方法も提案されている[7].

一方，*1.2* で述べた通り，Android は windows 等に比べると decompile が容易であり，元の高級言語への復元ツールが存在する．最近では既存のアプリケーションを改変して悪意のあるコードをインジェクトし，repackage するようなマルウェアが増えている[8]．理由として，

・リーバスエンジニアリングツールの充実化，

・パッキングや難読化といったソフトウェアプロテクションが適応されているケースが少ない．

といった原因が考えられる．

Dong 氏の大規模調査研究[9]によると， Class 名の変更によって，難読化を行っている割合は，GooglePlayStore で 43%，Third-Party Market で 73%，Malware で 63.5%と言われている．また Packing に関しては APK に適応している割合が，GooglePlayStore で 0.0%，Third-Party Markets で 10.4%，Malware で 2.1%という調査結果が出ている．この調査から他の OS のアプリケーションに比べてソフトウェアプロテクションがされている割合が少ないことがわかる．

我々はこのようなセカンダリタイプマルェアに対して有効的な検知方法を提案する．

3. 提案手法

前述の問題を解決するために，我々はコンパイラの違いに着目した検知方法を提案する．まず *dex* のヘッダー情報に着目して，コンパイラの検出を行い，その抽出した情報を bit 列へ変換し，パーミッション情報と共に複数の学習アルゴリズムへの入力パランメータとして利用した．

3.1 コンパイラの検出

コンパイラの検出には *dex* のヘッダー情報に着目した．以下の図 4 が *dex* のフォーマットである．

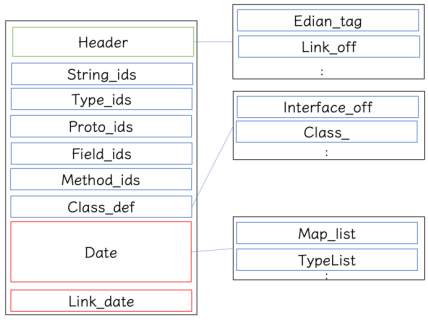

図 4. Dex format

Dex は Dalvik VM 上で動作するファイルであり，Windows でいう PE ファイルに該当する．*Dex* ファイルは非常にフレキシブルであり，コンパイラによって若干の違いが生じる．*1-2* で述べたように repackage の際には，Android Studio に実装されているコンパイラ(*dx)*ではなく，非標準的のコンパイラ(*dexlib, jack* など)が用いられることが多い．

Dex は大きく分けて，*Header*，*String_ids*，*Tpye_ids*，*Proto_ids*，*Field_ids*，*Method_ids*，*Class_def*，*Data*，*Link_data* に分けられている．*Data* 領域，*Link_data* 領域には可変長のデータが格納されており，それ以外の領域には固定長のデータが格納されている．我々はこの中でも，*Header, Class_def, data* に着目して検出した．以下が特徴である．

- *Data->Map_list* の配置が異なる
- *Class_def->internet_off* の値が，通常 interface を持っていない場合は 0 になるが，非標準的なツールでは null byte(null=10156)を値として与えている．
- *Endian_tag* が通常は Little Endian だが，非標準的なツールでは Big Endian になっている．
- *dx* では *header_size* は通常 0x70 byte になるが，非標準的なツールで作られた *dex* はそれ以外の値になる．
- *Link_offset* の値は通常 0byte になる．

以上のような違いがある．これらをシグネチャマッチングツールである *yara* を用いて検出を行った．

3.2 機械学習

本研究では，3-1 の方法コンパイラの種類を特定し，その情報を機械学習への適応を行った．使用したアルゴリズムは，以下の 5 種類であり，簡単な概要を以下に示す．

- SVM(Support Vector Machine)

この分類機は，教師あり学習の一つで，基本的に２クラス分類に用いられる．訓練サンプルから，各データテントの距離が最大となるように設定し，決定境界線を引く．それによりモデル自体の汎化性能の向上させる．

一般的に以下の目的関数の最小値の時の w の最大値を，二次計画法を用いて解く．

$$f = \frac{\omega^2}{\left(\omega^T(x_p - x_n)\right)} \tag{1}$$

訓練データから写像を生成して線形分離を容易にするカーネルSVCがあり，本研究では線形SVCを用いている．

- KNN

K 近傍(K-nearest neighbor algorithm)は特徴空間における最も近い訓練サンプル基づいた分類の手法であり，教師あり学習の一つである． 学習データをベクトル上へプロットし，未知のデータをそこから距離が近い順に任意に k 個取得し，多数決でデータが属するクラスを推定する．この時に基本的に，ユークリッド距離が使われる．探索アルゴリムは K-D Tree を用いており，計算量は以下の通りになる．N がデータ数, D が次元数.

$$\mathrm{O}\left(DNlog(N)\right) \tag{2}$$

- Random Forest

複数の決定木を多数生成することでモデルを作成を行う，アンサンブル学習である．決定木学習は，訓練データの集合要素についてばらつきを現象させることを企図して特徴量を分割を行う．

$$g = 1 - \sum_{i=1}^{K} p^2(C_i|t) \tag{3}$$

一般的な決定木学習は木が深くなるほ過学習に陥りやすいが, Random Forest では起きにくく，ジニ係数を見ることで特徴量の重要性を見ることができ，取捨選択の時にこれを用いることもある．

- Naïve Byes

ベイズの定理をもとにしており，以下の数式で表すことができる．

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \tag{4}$$

この時, P(A)は定数として見做すことができ, P(A|B)以下のように表せる．

$$P(A|B) = \prod_{i=1}^{N} P(x_i|Y) \tag{5}$$

また本研究では 0 と 1 のベルヌーイ分布を仮定している．

$$P(x;p) = q^x(1-q)^{1-x} \tag{6}$$

- Logistic Regression

ロジスティック回帰はベルヌーイ分布を用いた確率統計的分類機であり，予測変数をロジット変換することで確立分布にし，説明変数の式を探索する．活性化関数にはシグモイド関数を用いる．

$$\phi(z) = \frac{1}{1 + e^{-z}} \tag{7}$$

教師あり学習でも用いられる．

以上の 5 種類を使用し, *3-1* で抽出したコンパイラの情報を入力パラメータとして学習を行った

3.3 特徴量抽出

特徴量抽出に関しては以下のように行った．今回はパーミッション情報とコンパイラの種類を復習の学習アルゴリズムへの入力情報として取り入れた．図 5 に特徴量抽出から学習までのプロセスを示す．

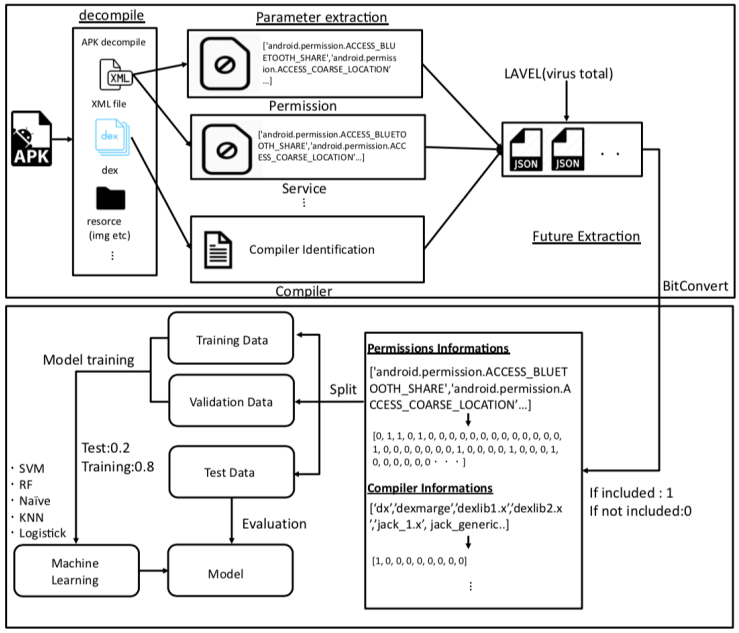

図 5. 学習プロセス

まず APK から *python3* と *Androgurard* を用いてパーミッション情報などを JSON へ変換する．この時にコンパイラの情報も同様に書き込む．次に変換した JSON を bit 列へ変換した．もし特定のパーミッションが含まれていたら 1, 含まれていなかったら 0 という形で bit 列で表現した．次にこれらをホールドアウト検証を用いて，ランダムに学習データとテストデータへ分割(8:2)を行い，機械学習の入力パラーメータとして入力し，これらを評価した．

4. 評価

今回はパーミッションのみの場合と，パーミッション情報とコンパイラ情報を組み合わせた時の，精度，F 値，誤検知率の比較を行った．Dataset，評価方法，評価結果に関して以下で述べる．

4.1 Dataset に関して

悪性サンプルは University of Luxembourg[10]から，入手したもので，図 6 の通り，VirusTotal でスキャンを行い 38 社以上のベンダーがマルウェアだと判断したサンプル 3000APK を使用した．良性サンプルも同様に，VirusTotal でスキャンを行った時にベンダーが一つも検知しなかった 3000APK を利用した．

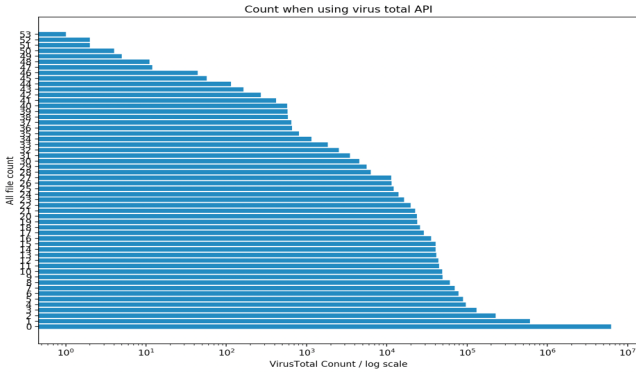

図 6. Virus Total Graph

図 7 が悪性サンプル 3000，良性サンプル 3000 のパーミッション情報を抽出した時の，使われていたパーミッションの上位 20 である．また図 7 は悪性の上位 20 を基準としている．赤が悪性，青が良性で用いられていたパーミッションの数の比較である．*READ_SMS* や *SEND_SMS*, *WRITE_CONTACTS* といったパーミッションは良性と比べてマルウェアがよく使われていることがわかる．

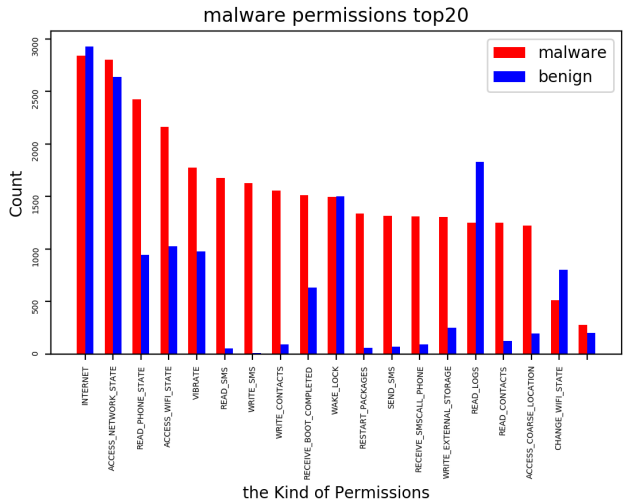

図 7. パーミッション比較

また，3-1 で述べた検出の結果を以下に示す．正規の APK は 9 割以上通常のコンパイラ dx でコンパイルされていること，マルウェアでは 7 割以上が非標準的なコンパイラでコンパイルされていることがわかった．

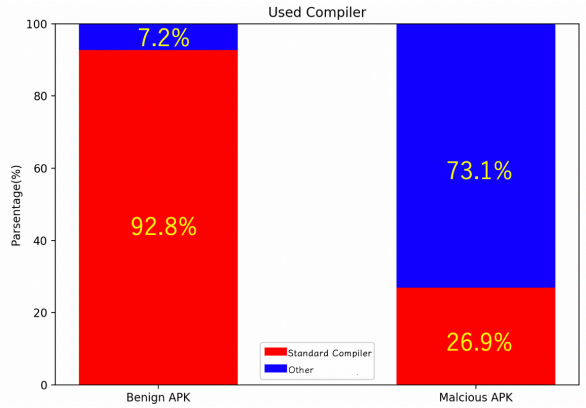

図 8.Compiler 比較

4.2 評価方法

我々はマルウェア評価に識別性能を用いた．識別性能は未知のマルウェアであるか否かを識別する能力の高さを示す指標である．まず検体は，真のラベル(正解ラベル)と *python3* と *Scikit-Learn* による識別結果(推定ラベル) 次に表 1 に示す混合行列の各区分に含まれる検体数を算出する．個別性を表す値としては F 値(以下: F-meature)と精度(以下: Accuracy)を用いた．

表 1. 混同行列

		推定ラベル	
		malware	benign
正解ラベル	malware	True Positive	False Negative
	benign	False Positive	True Negative

表 3 の混同行列に関して説明する．マルウェアであることを正しく検知した True-Positive(以下:*tp*)，マルウェアと予測したが実際はマルウェアではなかった False-Positive(以下:*fp*)，マルウェアではないと予測したが実際はマルウェアだった True-Negative(以下:*tn*)，マルウェアでないことを正しく判断した True-Negative(以下:*tn*)を基準とする精度と F 値で評価する．

ここで一般的な評価尺度として，再現率(以下:Recall)，適合率(以下:Precision)，精度(以下:Accuracy)，F 値(以下: F-measure)を使用した． Recall, Precision, Accuracy, F-measure はそれぞれ式(8),(9),(10),(11)で算出される．

真にマルウェアである識別検体の中で正しくマルウェアと推定された割合を再現率(Recall)とする．

$$Recall = \frac{tp}{tp + fn} \tag{8}$$

マルウェアと判定された検体の中で，真にマルウェアである検体の割合を適合率(Presiction)とする．

$$Precision = \frac{tp}{tp + fp} \tag{9}$$

F 値(F-meature)は適合率と再現率の調和平均を示す値である．

$$F - meature = \frac{2 * (Recall * Precision)}{Recall + Presicion} \tag{10}$$

精度(Accuracy)は予測ラベル全体と，正解ラベルがどれくらい一致しているかを判断する指標である．

$$Accuracy = \frac{tp + tn}{tp + fp + tn + fn} \tag{11}$$

F-meature, Accuracy は 0 以上 1 以下の値になり，1 に近いほど，手法の正確性が正しいことを意味する．本研究においては，マルウェアの検出性能評価に F 値と精度を用いる．

4.3 評価結果

今回は 10 回学習し平均を算出するという作業を 10 回行い，結果をグラフで表した．左のグラフがパーミッションのみで右がパーミッションとコンパイラ情報を組み合わせた結果である．図 9 に混同行列の結果を示す．

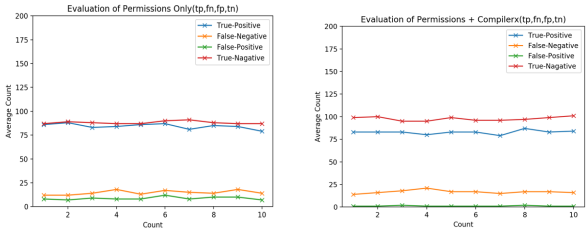

図 9.　混同行列

図 10 に Accuracy の結果を示す

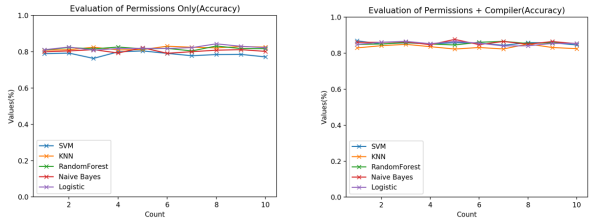

図 10. Accuracy 結果

図 11 に F 値の結果を示す．

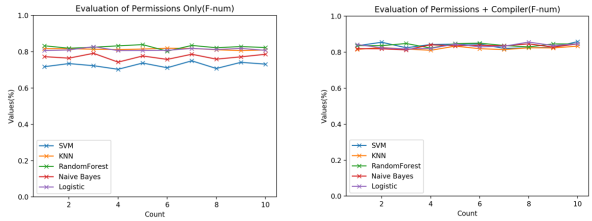

図 11. F-meature

表 2. 判定結果

	Accuracy	F-meature	誤検知率
Permission	79.25%	77.45	4.35
Permission + Compiler	84.11%	83.55%	0.43%(1%未満)

表 2 が判定結果のまとめである．パーミッションのみと比べると，コパイラ情報を組み合わせることで，F 値，Accuracy 共に全体的に 5%ほど精度が向上していることがわかる．また誤検知率に関しても，パーミッションのみの場合 5%ほどあるが，コンパイラ情報を組み合わせることで，1%未満まで誤検知率を抑えることができた．

しかし，アルゴリズムによっては検知率が若干下がることをも確認できた．そのため，適切なアルゴリズムを検討する必要がある．

5. 結論

本研究ではコンパイラの情報を特定し，それを機械学習へ適応，検知，評価を行った．本研究によって，通常のパーミッションベースの検知法に組み合わせることで，誤検知，精度，F 値が向上することが確認できた．しかし，Packing などのソフトウェアプロテクションが施されている場合，DEX の構造が変化するので，今後はコンパイラの情報だけではなく，Packer の情報と Packing されて DEX の復元し，Dex のコンパイラ情報を抽出することを行いたい．また パーミッション情報だけではなく，API コール，バイナリ情報といった様々な情報に適応を行い，相関に関しての分析等行いたいと考えている．

参考文献

[1] “情報電子機器の保有所有率”, http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h30/html/nd252110.html, 総務省, 2018 年.

[2] Tam Kimberly et al., "The evolution of android malware and android analysis techniques", *ACM Computing Surveys*, 2017.

[3] “APKTOOL”. https://ibotpeaches.github.io/Apktool/install/

[4] “Dex2jar”, https://github.com/pxb1988/dex2jar

[5] Wei-Ling, Hung-Min Sun, Wei Wu, “An Android Behavior-Based Malware Detection Method using Machine Learning”, ICSPCC2016.

[6] Xiangyu-Ju, “Android malware detection though Permission and Package”, Processings of the 2014 International Conference on Wavelet Analyssi and Pattern Recognition, Lanzhou, 14-16 july, 2014.

[7] Mahima Choudhary, Brij Kishore, “HAADM: Hybrid Analaysis for Android Malware Detection”, 2018 International Conference on Computer Communication and Informatics(ICCCI-2018), India, 2018.

[8] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri, “A study of Android Application Security”, Proceeding of the 20th USENIX Security Symposium, 2011

[9] Dong S. et al. (2018) Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. In: Beyah R., Chang B., Li Y., Zhu S. (eds) Security and Privacy in Communication Networks. SecureComm 2018. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 254. Springer, Cham

[10] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. Mining Software Repositories (MSR) 2016.