一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2017-95(2018-02)

正当な DEX ファイルの生成による Android 仮想マシンのランダムテスト

池尾　弘史†　清水遼太朗†　石浦菜岐佐†

† 関西学院大学 理工学部　〒 669–1337 兵庫県三田市学園 2–1

あらまし　本稿では, 正当な DEX (Dalvik Executable) ファイルをランダムに生成することによって Android 仮想マシンをテストする手法を提案する. Android 仮想マシンのランダムテスト手法としては, 既存の DEX ファイルを変異させることによりテストケースを生成する手法と, ランダムに生成した Java プログラムをコンパイルして得られる DEX ファイルを用いる手法がある. 前者には仮想マシンのベリファイアを通過できないテストケースを多く生成してしまうという課題があり, 後者には生成されるバイトコードのパターンがコンパイラのコード生成ポリシーにより限定されてしまうという課題がある. 本稿の手法では, ベリファイアを通過する 正当な DEX ファイルをコンパイラを用いず直接生成する. ランダムな命令列および計算結果の照合を行うバイトコードを生成し, DEX ファイルの各セクションの内容を生成したバイトコードに応じて修正することにより, 必ずベリファイアを通過する DEX ファイルを生成する. 本手法に基づくテストシステムを Perl で実装し, Android 6.0 および 8.0 の仮想マシンを対象にテストを行った結果, 不具合を検出することはできなかったが, ベリファイアを 100% 通過し, コンパイラが生成しにくい命令列を含む DEX ファイルを生成することができた.

キーワード　Android 仮想マシン, コンパイラ, ランダムテスト, ファジング

Random Testing of Android Virtual Machine by Valid Dex File Generation

Hirofumi IKEO†, Ryotaro SHIMIZU†, and Nagisa ISHIURA†

† Kwansei Gakuin University,　Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan

Abstract　This article presents a method of testing Android virtual machines (VMs) by generating random but valid DEX (Dalvik Executable) files. There have been two major methods of generating random DEX files to test Android VMs; one is to randomly mutate bits in existing DEX files and the other is to generate random Java programs and to compile them. While the former approach ends up with generating many invalid DEX files that are rejected by the VM's verifier, the latter can generate limited patterns of VM machine codes that follow the compilers' code generation policies. The method proposed in this article randomly generates only valid DEX files without using Java compilers. Bytecode sequences to compute random arithmetic operations and to verify their results are first generated, and then they are embedded into a template file and the contents of all the section in the file are adjusted so that the DEX file will be valid. A test system has been implemented in Perl 5 and the virtual machines of Android 6.0 and 8.0 have been tested. 100% of the generated files passed the verifiers and the bytecodes exhibited different register access patterns than those generated by front-end compilers, though no error has been detected so far.

Key words　Android Virtual Machine, Compiler, Random Tesing, Fuzzing

1. はじめに

Android (注1) はスマートフォンやタブレット等のモバイル端末向けに開発された OS であるが, 最近では組込みシステムにも搭載される等, Android が活用される局面が増えてきている. 2017 年には月間のユーザー数が 20 億人を突破し, 今もそのユーザー数を増やし続けている (注2). このような利用の拡大に

(注1)：https://source.android.com (accessed 2018-02-05)
(注2)：https://techcrunch.com/201705/17/google-has-2-billion-users-on-android-500m-on-google-photos/ (accessed 2018-02-05)

伴い, Android の信頼性確保は社会的にも非常に重要な課題となってきている.

Android は カーネルやライブラリ, Android ランタイム (仮想マシンとコアライブラリ), アプリケーションフレームワーク, および各種アプリケーション等様々なソフトウェアで構成され, それぞれがテスト対象となるが, 本稿では仮想マシンのテストを対象とする. 仮想マシンは高性能化や機能追加のために仕様や実装の変更が頻繁に行われるため, その分不具合が発生し易く, そのテストは重要である.

仮想マシンのテストは, まず開発者が作成したテストケースを用いて徹底的に行われる. しかし, これらを経てもなお不具合が潜在する場合があり, そのような不具合を検出するために, ランダムに生成したプログラムによりテストを行うランダムテスト (ファジング) が用いられる.

Dexfuzz [1] は, DEX ファイル (Android 仮想マシンの実行可能ファイル) の変異体 (ファイル中のいくつかのビットを反転させたもの) を多量に生成することにより Android 仮想マシンのテストを行っている. 変異に基づくファジングでは, ベリファイア (不正な DEX ファイルを検出するモジュール) をいかに通過させるかが課題になるが, Dexfuzz は命令の構文を解析した上で特定の箇所を変異させることにより, ベリファイアの通過率を向上させている. しかし, この手法によっても DEX ファイルのベリファイア通過率は 10% 程にとどまる. また, 差分テスト手法 [2] を用いているため, DEX ファイルの実行結果が仕様通りであるかを検証できないことも課題である.

文献 [3] では, ランダムに生成した Java プログラムからフロントエンドコンパイラ (javac および dx) を用いて DEX ファイルを生成することにより仮想マシンのテストを行っている. しかし, この DEX ファイルはコンパイラが生成するため, オペランドのレジスタ番号の範囲や命令等の種類がコンパイラのコード生成ポリシーによって限定されるという課題がある.

本稿では, ランダムでかつ正当な DEX ファイルを直接生成することによって Android 仮想マシンのテストを行う手法を提案する. 本手法では, ランダムな命令列とその命令列の実行結果と期待値 (期待される実行結果) の照合を行う DEX のバイトコードを生成する. また, バイトコードの内容に応じて DEX ファイルの各セクションの内容を修正することにより, 必ずベリファイアを通過する DEX ファイルを生成する.

以下, 本稿では 2 章で ART の構造と DEX ファイル, および仮想マシンのテスト手法について述べ, 3 章では本手法に基づくシステム及び DEX ファイルの生成手法について述べる. 4 章で実装及び実験について述べた後, 5 章でまとめと今後の課題について述べる.

2. 仮想マシンのファジング

2.1 Android 仮想マシン ART の構成

ART [4] は Android 5 (Lolipop) 以降で採用されている仮想マシン (Virtual Machine; VM) である. それ以前の Dalvik VM [5] がインタプリタと JIT (Just-In-Time) コンパイラを組み合わせていたのに対し, ART は, 実行効率向上のためにインタプリタと AOT (Ahead-Of-Time) コンパイラを用いて最適化を強化している.

ART の構成の概略 [6] を図 1 に示す. DEX ファイルは Android 仮想マシンの実行可能形式であり, バイトコード (命令列) を含んでいる. ART は DEX ファイルを受け取ると, dex2oat

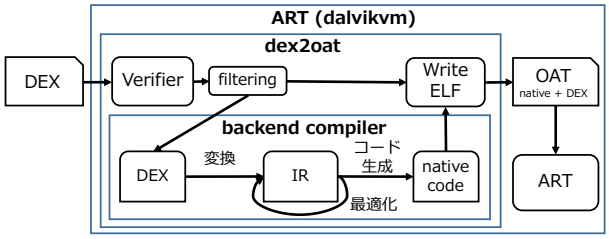


図 1: ART の構成 [6]

により検証とコンパイルを行って, 元の DEX ファイルの内容にターゲットプロセッサ用のネイティブコードを加えた ELF 形式の実行可能ファイル OAT を生成し, それを ART の仮想マシンで実行する.

dex2oat は初めにベリファイア (Verifier) で入力された DEX ファイルの正当性の検査を行う. ベリファイアはチェックサムやファイルフォーマットの検査のみならず, 各セクションが持つオフセットの先にきちんとデータが存在しているか, レジスタ番号がバイトコードで使用可能な範囲に収まっているか, 分岐命令のジャンプ先 (オフセット) がバイトコードの範囲内で, それが命令の先頭を指しているか等, バイトコードの静的な正当性まで検査する.

次のフィルタリングでは, バイトコードからネイティブコードにコンパイルする箇所を抽出する. 即ち, ART では全てのバイトコードではなく, メソッドの規模 (命令数あるいは使用レジスタ番号の最大値) 等が一定の条件を満たしたものだけがネイティブコードにコンパイルされる. バックエンドコンパイラ (backend compiler) は, コンパイルする部分を中間表現 (IR) に変換して最適化を行った後, IR からネイティブコードを生成する.

2.2 DEX ファイルの構造

DEX ファイルはバイトコード命令列の他, 情報の保存位置のポインタ, プログラム内で使用される型の情報, 定数文字列データ, 初期値データ等を持つ. DEX ファイルのフォーマットを図 2 に示す. Data セクションには, バイトコードや文字列データ, クラスデータ, フィールド情報等が格納されている. Header セクションにはチェックサムや署名, ファイルサイズ, 各セクションへのオフセット等の情報が格納されている. String_ids, Type_ids, Proto_ids, Field_ids, Method_ids セクションにはそれぞれのデータのインデックス, Class_defs セクションにはクラスデータ (クラスフィールドの修飾子等) へのオフセット等が格納されている.

2.3 DEX バイトコード

Android 仮想マシンはレジスタマシンであり, メソッド毎に最大 65536 個の 32 ビットレジスタ (0, 1, 2, ··· と番号付けされる) を持つことができる. 命令は 3 アドレス形式であり, 例えば “`add-int v3, v5, v6`” は 5 番レジスタと 6 番レジスタの値を 32 ビット整数として加算した結果を 3 番レジスタに格納する命令である.

命令がオペランドに指定できるレジスタの番号の範囲は命令毎に異なり, 0~15, 0~255, もしくは 0~65535 のいずれかである. 算術命令の多くが 0~255 の範囲を用いているため, 256 番以降に格納された値に対して算術演算を行う場合には, 255 番以前のレジスタに値を移動させなければならない.

int 型等 32 ビット以下の値は 1 つのレジスタに格納されるが, long 型等の 64 ビット型の値はレジスタペア (番号が連続する 2 つのレジスタ) に格納され, 命令のオペランドにはレジ

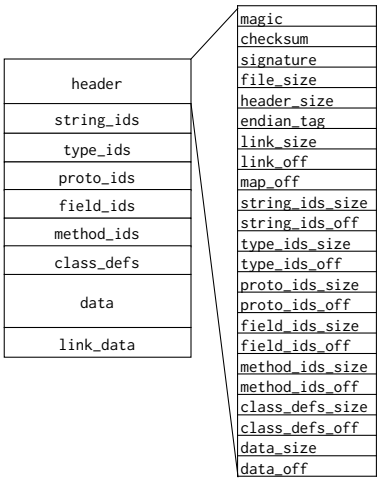


図 2: DEX ファイルの構造

スタペアの若い方のレジスタ番号を指定する. 64 ビットのデータを書き込んだレジスタペアのいずれかに 32 ビットデータを操作する命令でアクセスがあると, フォーマット違反になり, ベリファイアを通過できなくなる.

クラス内のメソッド間で変数を共有する場合はクラスフィールドと呼ばれる領域を用いる. クラスフィールドには `sput-`*typ*, `sget-`*typ* (*typ* はデータの型) を用いてアクセスする. `sput-`*typ* はクラスフィールドに *typ* 型の値を設定する命令であり, `sget-`*typ* はクラスフィールドから *typ* 型の値を取得する命令である.

2.4 仮想マシンのファジング

ファジングとは, 開発者が予期していないような入力を与えてソフトウェアをテストする手法である [7]. ファジングは, 既存の入力を変異させて新たな入力を生成する手法と, 一から入力を生成する手法に分類される. 前者は実装が比較的容易だが, 不正な入力を生成しやすいためテスト効率が下がることがある. 後者は, テスト対象のプロトコルやファイルの仕様に基づいて入力を生成するものであり, 前者よりも高い割合で正当な入力を生成できるが, 入力形式の理解や生成系の実装にコストがかかる.

Java 仮想マシンのテストに関しては文献 [8] [9] [3] がある. 文献 [8] は正当なクラスファイルを生成することにより仮想マシンの JIT (Just In Time) 機能をテストしている. 文献 [9] は, 既存のクラスファイルを変異させコードカバレッジの高い入力を生成し, 様々な実装の Java 仮想マシンで実行する差分テストを行っている. 文献 [3] では, ランダムテストシステム [10] で生成した Java プログラムをコンパイル, 実行することにより仮想マシンを含む Java 処理系をテストしている.

Android 仮想マシンのテストに関しては [1] [3] がある. 文献 [1] では, 既存の DEX ファイルを変異させる方法によりファジングを行っている. ベリファイアの通過率を向上させるために, 既存の DEX ファイルを解析し, オペランドレジスタの番号や条件分岐のオフセット等のフィールドを特定した上で変異させている. しかし, フォーマットに従って値を変異させた場合でも, その値がバイトコードにおいて無効なものであればベリファイアが検出してしまう. 例えば, レジスタ番号がそのメソッドで使用可能なレジスタの範囲を超えた場合や, 分岐命令のジャンプ先がバイトコードの範囲外になった場合等である. また, この手法では内部コンパイラの 3 種類のオプションでコンパイル · 実行して得られる結果を比較し, 多数決でエラー判定を行っている. しかし, すべてのオプションで同じ誤りが起こった場合には見逃しが生じてしまうため, 実行結果が正しいかどうかは厳密にテストできていない.

文献 [3] では, ランダムに生成した Java プログラムからフロントエンドコンパイラ (javac および dx) を用いて DEX ファイルを生成することにより, これらのコンパイラと ART の処理系のテストを行っている. この手法では, 必ず正当な DEX ファイルを生成できるが, 使用されるレジスタが若い番号に偏ってしまったり, 命令を選択する規則が一定である等, バイトコードの内容がフロントエンドコンパイラの仕様により限定される. このため, テスト数をいくら増やしても, 仮想マシンにテストされない命令やオペランドの組み合わせが残ってしまうという課題がある.

3. 正当な DEX ファイル生成による ART のランダムテスト

3.1 概　　要

本稿では, 正当な DEX ファイルをランダムに生成することにより Android 仮想マシンをテストを行う手法を提案する. テスト対象は, 図 1 のベリファイア, バックエンドコンパイラ, および ART である. 本手法で生成する DEX ファイルベリファイアに妨げられることなくコンパイラや ART をテストできる. また, 本手法ではフロントエンドコンパイラでは生成されにくい命令列を含む DEX ファイルを生成することができる.

本手法で生成する DEX ファイルのバイトコードの例 (主要部分の逆アセンブル結果) を図 3 に示す. 図 3 (c) の main は, (1) 変数定義ブロック (`0001`∼`1135` 行目), (2) 算術計算ブロック (`1136`∼`4257` 行目), (3) 期待値照合ブロック (`4258`∼`4484` 行目) より構成されている.

本手法では, Java プログラムをコンパイルすることによって得られた DEX ファイルのテンプレートに (1)∼(3) の命令列を生成し挿入した後, ヘッダー等のセクションをその命令列に応じて修正することにより, ベリファイアで不正と判定されない DEX ファイルを生成する.

3.2 命令列の生成

本手法では, まず計算の結果どのような値がレジスタに入るかを先に決定し, そのような値を生成する命令列を実行の逆順に決定していくことにより命令列を生成する.

1. 期待値照合ブロックの生成

まず, (3) での期待値照合の対象となるレジスタをランダムに選択し, そのレジスタの期待値をランダムに決定する. 次に, これらのレジスタの値と期待値との一致判定を行う命令列を生成する. 図 3 (c) では, 計算の結果を格納するレジスタとして, レジスタ `v4373`, `v1697`, および `v3603`∼`3604` が選ばれ, それぞれの期待値として `0xcfe1311b`, `0x124615e5`, `0x6e5f5a2d68124931` が選ばれている. `4261`∼`4266` 行では, `v4373` レジスタの値が `0xcfe1311b` になっているかどうかテストしている. `v6` に期待値を, `v8` に `v4373` の値をロードし, それらの比較に基づいて条件分岐している. 等しい場合には, `OK` を出力し, そうでない場合には `NG` を出力している.

2. 算術計算ブロックの生成

1. で選択したレジスタに期待値通りの値を格納するような命令列を生成する. これは下記の手順による命令生成を実行の逆順に繰り返すことにより行う.

```
[000240] seed.<clinit>:()V
0001: const-wide v0, 0xc3fa49d96b8b20e6
0002: sput-wide v0, Lseed;.i5:I
...
0011: const v0, 0xfffff251
0012: sput v0, Lseed;.i0:I
0013: return-void
```

(a) clinit

```
[00029c] seed.<init>:()V
0001: move-object v0, v2
0002: move-object v1, v0
0003: invoke-direct {v1}, Ljava/lang/Object;.<init>:()V // main
0004: return-void
```

(b) init

```
[0002b8] seed.main:([Ljava/lang/String;)V

// (1) 変数定義ブロック
0001: sget-wide v8, Lseed;.i5:I
0002: move-wide/16 v8681, v8
...
1134: const-wide v8, 0x0000000000017e74
1135: move-wide/16 v5793, v8

// (2) 算術計算ブロック
1136: move-wide/16 v12, v14435
1137: move-wide/16 v14, v5502
1138: or-long v10, v12, v14
1139: move-wide/16 v14655, v10
...
4254: move-wide/16 v12, v12967
4255: move-wide/16 v14, v15600
4256: div-long v10, v12, v14
4257: move-wide/16 v12862, v10

// (3) 期待値計算ブロック
4258: sget-object v0,
        Ljava/lang/System;.out:Ljava/io/PrintStream;
4259: const-string v2, "OK"
4260: const-string v3, "NG"

4261: const v6, 0xcfe1311b
4262: move/16 v8, v4373
4263: if-ne v8, v6, +0006
4264: invoke-virtual {v0, v2},
       Ljava/io/PrintStream;.print:(Ljava/lang/String;)V
4265: goto +0004
4266: invoke-virtual {v0, v3},
        Ljava/io/PrintStream;.print:(Ljava/lang/String;)V

4267: const v6, 0x124615e5
4268: move/16 v8, v1697
4269: if-ne v8, v6, +0006
4270: invoke-virtual {v0, v2},
        Ljava/io/PrintStream;.print:(Ljava/lang/String;)V
4271: goto +0004
4272: invoke-virtual {v0, v3},
        Ljava/io/PrintStream;.print:(Ljava/lang/String;)V
...
4478: const-wide v6, 0x6e5f5a2d68124931
4479: move-wide/16 v8, v3603
4480: cmp-long v4, v6, v8
4481: if-nez v4, +0006
4482: invoke-virtual {v0, v2},
        Ljava/io/PrintStream;.print:(Ljava/lang/String;)V
4483: goto +0004
4484: invoke-virtual {v0, v3},
        Ljava/io/PrintStream;.print:(Ljava/lang/String;)V

4485: return-void
```

(c) main

図 3: 生成するバイトコード例

(a) 値を書き込むレジスタをランダムに一つ選択する.

(b) 使用する演算とそのオペランドの型と値を決定する.

(c) オペランドレジスタを選択する. 一定の確率でレジスタの再利用を試みる.

- 型と値が (b) で決定したものに一致するレジスタがあれば, 一定の確率でそれを使用する (再利用).
- それ以外の場合には, 新しいレジスタをランダムに選択し, そのレジスタに (b) で決定した値を設定する.

本手法による命令列の生成例を図 4 に示す. この例では 1. の操作によりレジスタ v16 と v18 が選ばれ, その期待値がそれぞれ 1 と 3 と設定される (①). 次に 2-(a) の操作でレジスタ v16 がもつ int 型の 1 が選ばれ, 演算結果がその値となるように, 2-(b) の操作で, sub-int (減算) 演算とそのオペランドとして int 型の 4 と int 型の 3 が決定される. 2-(c) の操作で int 型の 3 の値を持つ v18 が選ばれ, 新しく v19 に 4 が設定される (②).

更に, ② に対して 2. の操作を行い, レジスタ v18 がもつ int 型の 3 が選ばれ, add-int (加算) 演算と int 型の 3 と int 型の 0 が決定される. それらのオペランドがレジスタ v16 と v17 に設定される (③).

本手法では任意の番号のレジスタを用いた命令列を生成することができる. 命令で使用できるレジスタに制約がある場合には, レジスタの値を転送するための命令列も生成する. 例えば, 図 3 (c) の算術計算ブロックでは, 1136∼1137 行目でオペランドとして使用する値をそれぞれ v14435, v5502 から v12, v14 に, v10 に格納された演算結果を v14655 に転送している.

レジスタペアの一方のみを書き換える命令はフォーマット違反となる. また, レジスタにはオブジェクトへのリファレンス等の算術演算で扱わないデータが格納されることもあるため, 本手法では展開の際に, それらを不正に扱うことがないようにレジスタの内容に基づき制御を行う.

3. 変数定義ブロックの生成

2. で必要になる初期値をレジスタに設定する命令列を生成する. 図 4 の例において, (2) の操作を ③ で終了する場合, v16, v17, v19 をそれぞれ 0, 3, 4 で初期化する必要がある. これらの初期値は, const 命令 (レジスタに定数をロードする) または sget 命令 (レジスタにクラスフィールドが持つ値をロードする) によりレジスタに設定する. クラスフィールドの値を使用する場合は, (a) の clinit で初期値を設定する. (c) の main 0002∼0003 行目ではレジスタ v17 と v19 に定数 3, 4 をロードしている. (a) の clinit 0001∼0002 行目で, 0 をクラスフィールドに初期値を設定し, (c) の main 0001 行目でその値をレジスタ v16 にロードしている.

本手法で生成する DEX バイトコードの諸元を表 1 に示す. 変数に関しては, 型は 32, 64 ビットの符号付き整数型, スコープはローカルおよびクラスフィールド, 修飾子は static (クラスフィールド) と final (再代入の禁止) および volatile (スレッド間で変数の値を同期) の組み合わせを用いる. 命令は主に演算命令と型変換命令を対象とする.

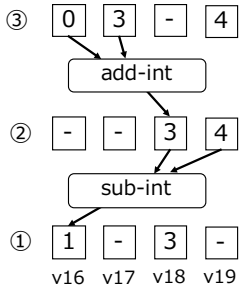
3.3 DEX ファイルの生成

本手法における DEX ファイルの生成は以下の手順で行う.

1. テンプレートとなる DEX ファイルを用意する.
2. 3.2 節の手法で生成した命令列をテンプレートの Data セクションに挿入する.

表 1: 本手法で生成する DEX バイトコードの諸元

型	int (32 ビット), long (64 ビット)
スコープ	ローカル, クラスフィールド
修飾子	static, static final, static volatile
演算命令	add-int, sub-int, mul-int, div-int, rem-int, and-int, or-int, xor-int, shl-int, shr-int, ushr-int, long-to-int, add-long, sub-long, mul-long, div-long, rem-long, and-long, or-long, xor-long, shl-long, shr-long, ushr-long, int-to-long



(1) 命令列生成の流れ

```
0001: const v0, 0x0
0002: sput v0, Lseed;.i0:I
0003: return-void
```

(a) clinit

```
0001: move-object v0, v2
0002: move-object v1, v0
0003: invoke-direct {v1}, Ljava/lang/Object;.<init>:()V // main
0004: return-void
```

(b) init

```
0001: sget v16, Lseed;.i0:I
0002: const v17, 0x3
0003: const v19, 0x4
0004:
0005: add-int v18, v17, v16
0006: sub-int v16, v19, v18
0007:
0008: sget-object v0, Ljava/lang/System;.out:Ljava/io/PrintStream;
0009: const-string v2, "OK"
0010: const-string v3, "NG"
0011:
0012: const v6, 0x1
0013: move/16 v8, v16
0014: if-ne v8, v6, +0006
0015: invoke-virtual {v0, v2}, Ljava/io/PrintStream;.print:(Ljava/lang/String;)V
0016: goto +0004
0017: invoke-virtual {v0, v3}, Ljava/io/PrintStream;.print:(Ljava/lang/String;)V
0018:
0019: const v6, 0x3
0020: move/16 v8, v18
0021: if-ne v8, v6, +0006
0022: invoke-virtual {v0, v2}, Ljava/io/PrintStream;.print:(Ljava/lang/String;)V
0023: goto +0004
0024: invoke-virtual {v0, v3}, Ljava/io/PrintStream;.print:(Ljava/lang/String;)V
0025: return-void
```

(c) main

(2) バイトコード

図 4: バイトコードの生成

3. 命令列の内容に応じて各セクションの情報を修正する.

テンプレートとなる DEX ファイルは図 5 に示す Java プログラムをコンパイルすることにより得る. クラスフィールドの定義や定数文字列の出力を行う Java プログラムから, それらの情報 (OK, NG の定数文字列データ, プリント出力に必要なオブジェクト等) を各セクションに格納した DEX ファイルを用意する.

この Java プログラムから生成される DEX ファイルのテンプレートは, 図 5 の 0003∼0024 行目で定義している 20 個のクラスフィールドの情報, 0027∼0028 行目の定数文字列の出力に必要な情報を保持している. 本手法ではこれを利用して最大 20 個のクラスフィールドを持つ変数定義ブロック (1) を生成する.

生成した命令列の挿入は, テンプレートがもつ命令列を上書きする形で行う. 命令列の挿入により, データ等のオフセットが変化したり, メソッドのサイズや使用するレジスタの数が変化するので, 挿入した命令列の内容に合わせて各セクションの情報を修正する.

この方法で生成した DEX ファイルは必ずベリファイアを通過する.

```
0001: class seed {
0002:
0003:     // int vars
0004:     static int i0 = 0;
0005:     static int i1 = 0;
0006:     static int i2 = 0;
0007:     final static int i3 = 0;
0008:     final static int i4 = 0;
0009:     final static int i5 = 0;
0010:     volatile static int i6 = 0;
0011:     volatile static int i7 = 0;
0012:     volatile static int i8 = 0;
0012:     volatile static int i9 = 0;
0013:
0014:     // long vars
0015:     static long l0 = 0;
0016:     static long l1 = 0;
0017:     static long l2 = 0;
0018:     final static long l3 = 0;
0019:     final static long l4 = 0;
0020:     final static long l5 = 0;
0021:     volatile static long l6 = 0;
0022:     volatile static long l7 = 0;
0023:     volatile static long l8 = 0;
0024:     volatile static long l9 = 0;
0025:
0026:     public static void main(String[] args){
0027:         System.out.print("OK\n");
0028:         System.out.print("NG\n");
0029:     }
0030: }
```

図 5: テンプレート生成用 Java プログラム

4. 実 験 結 果

提案手法に基づくテストシステムを Perl 5.24.1 で実装した. 本システムは Mac OS, Ubuntu Linux 等で動作する.

表 2 に示す Android 8.0 (x86_64) と Android 6.0 (ARM) で動作する 2 つの仮想マシンに対して, 本手法によるランダムテストを適用した. 表 2 における “CPU” はターゲット, “Android” はバージョン, “VM” は ART のバージョン, “Option” は dex2oat の最適化オプションである.

この実験において生成する DEX ファイルは, 16380 個 (つまり v0∼v16379 レジスタ) のレジスタを使用し, そのうち 30

表 2: 実験対象

	CPU	Android	VM	Option
A	x86_64	8.0	ART 2.1.0	Optimizing
B	ARM	6.0	ART 2.1.0	Optimizing

表 3: テスト結果

(A) x86_64 (emulator)

通過率	時間 [h]	検出エラー数			
		verify	quicken	speed	everything
100.0%	25.1	0	0	0	0

(B) ARM (Nexus 7 2013)

通過率	時間 [h]	検出エラー数			
		interpret-only	balanced	speed	everything
100.0%	31.7	0	0	0	0

テスト数: 10,000
CPU: Intel Core i3-4010U CPU @1.70GHz x 4

表 4: レジスタアクセスの比較

	レジスタ数	レジスタ番号	
		最小値	最大値
文献 [3] の手法	345.6	0.0	490.0
本手法	1197.8	0.0	16367.9

個のレジスタについて期待値照合を行った. また, 各 DEX ファイルについて命令の生成の操作は 800 回行った.

実験は Ubuntu (Intel Core i3-4010U CPU @1.70GHz x 4) で行った. テスト生成部が Ubuntu 上で生成した DEX ファイルを, adb (Android Debug Bridge) ツールを用いてそれぞれのターゲットに転送し, 実行した. テスト対象の Android 8.0 (x86_64) は本システムと同じ Ubuntu 上で動作させた Android エミュレータ, 6.0 (ARM) は タブレット端末の Nexus 7 の 2013 モデル (Qualcomm Snapdragon S4 Pro 8064 クアッドコア, 1.5 GHz) を使用した.

実験の結果を表 3 に示す. “通過率” はテストした DEX ファイルのうちベリファイアを通過した割合, “時間” はテストに要した時間, “検出エラー数” はエラーを検出した数でフィルタリングオプション毎に示している. どちらの条件でもエラーを検出することはできなかったが, 提案手法で生成した DEX のベリファイアの通過率は 100 % であった.

本手法と文献 [3] で生成されるバイトコードのレジスタアクセスの比較を表 4 に示す. “レジスタ数” は DEX ファイル内で使用するレジスタの数の平均, “レジスタ番号” はバイトコードで使用されたレジスタ番号の範囲 (レジスタ番号の最小値および最大値の平均) である. 生成した DEX ファイルの数はそれぞれ 50 個であり, バイトコードの規模は 800 演算程度である. 文献 [3] の手法に比べ本手法は, 広範囲かつ多くのレジスタにアクセスするテストケースを生成できている.

エラーを検出できなかった一因として, 本手法で生成できる DEX ファイルのバイトコードの規模が小さいことが挙げられる. dex2oat はメソッド毎に使用するレジスタの数およびバイトコードの長さが一定以上の場合, ネイティブコードへの変換 (および最適化) を行わない. 高度なコンパイラは最適化機能に不具合を含むことが多い [11] が, これをテストするのに十分な規模のバイトコードがこの制約のために生成できていない. これを解決するためには, 複数のメソッドやクラス, インスタンスを組み合わせて複雑なバイトコードを生成することが必要と考えられる.

また, 生成するプログラム自体にも改良の余地がある. 本手法では, 使用するレジスタ番号も, オペランドの値も使用可能な範囲から単純にランダムに決定している. Orange4 [12] では境界値の比重を大きくして値を選択することによりコンパイラの最適化の不具合の検出能力が向上することが報告されているため, このような値選択法を採用する必要があると考えられる. また, 動的にクラスフィールドを生成し修飾子を組み合わせたり, 条件分岐やループ等の構文的要素を追加する等により, バイトコードをより複雑にする工夫も必要と考えられる.

5. むすび

本稿では, 正当な DEX を生成し, Android 仮想マシンのランダムテストを行う手法を提案した. 実験の結果, Android 6.0 および 8.0 で動作する ART において, 不具合を検出することができなかったが, 必ずベリファイアを通過する DEX ファイルを生成することができた.

今後の課題としては, 複数のクラス, インスタンス, メソッドを用いたバイトコードの規模の拡大, クラスフィールドの要素の動的な生成や構文的要素の追加等による複雑な DEX ファイルの生成が挙げられる.

謝辞

本研究に関してご協力, ご討議頂いた関西学院大学理工学部石浦研究室の諸氏に感謝致します.

文　献

[1] S. C. Kyle, H. Leather, B. Franke, D. Butcher, and S. Monteith: “Application of Domain-aware Binary Fuzzing to Aid Android Virtual Machine Testing,” in *Proc. VEE 2015*, pp. 121–132 (Mar. 2015).

[2] W. M. McKeeman: “Differential Testing for Software,” *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107 (Dec. 1998).

[3] 清水遼太朗, 池尾弘史, 石浦菜岐佐: “コンパイラのランダムテストシステム Orange3 の拡張による Java 処理系のテスト,” 信学ソ大, A-6-11 (Sept. 2016).

[4] Android Open Source Project (Android core technologies), `https://source.android.com/devices/tech/dalvik/` (accessed 2018-02-05).

[5] 松永達也: Android の仮想マシン Dalvik 編, 達人出版 (2015).

[6] M. Backes, S. Bugiel, O. Schranz, P. Styp-Rekowsky and S. Weisgerber: “ARTist: The Android Runtime Instrumentation and Security Toolkit” in *Proc. EuroS&P 2017*, pp. 481–495 (Apr. 2017).

[7] M. Sutton, A. Greene, and P. Amini 著, 伊藤裕之訳: ファジング ブルートフォースによる脆弱性発見手法, 毎日コミュニケーションズ (2008).

[8] T. Yoshikawa, K. Shimura, and T. Ozawa: “Random Program Generator for Java JIT Compiler Test System,” in *Proc. Quality Software*, pp. 20–23 (Nov. 2003).

[9] Y. Chen, T. Su, C. Sun, and J. Zhao: “Coverage-directed Differential Testing of JVM Implementations,” in *Proc. PLDI 2016*, pp. 85–99 (June 2016).

[10] E. Nagai, A. Hashimoto, and N. Ishiura: “Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Wxpressions,” *IPSJ TSLDM*, vol. 7, pp. 91–100 (Aug. 2014).

[11] 石浦菜岐佐: “コンパイラのファジング,” 電子情報通信学会 Fundamentals Review, vol. 9, no. 3, pp. 188–196 (Jan. 2016).

[12] K. Nakamura and N. Ishiura: “Random Testing of C Compilers Based on Test Program Generation by Equivalence Transformation,” in *Proc. APCCAS 2016*, pp. 676–679 (Oct. 2016).