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
VLD2018-124,HWS2018-87(2019-02)

Android 仮想マシンのランダムテスト における命令列生成の強化

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, Android 仮想マシンのランダムテストにおける命令列の生成を強化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において, 既存の DEX ファイルを変異させて生成する手法では, 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る. また, ベリファイアを通過できる DEX ファイルを生成する手法では, 1 ファイル中に生成できる命令数, 命令種類, およびクラスフィールドのバリエーションが少ない等の課題がある. 本稿では, 後者の Android 仮想マシンのテストシステムを拡張し, バイトコードの規模とバリエーションを拡大する. 本手法では, 複数のメソッドを生成することにより, 1 ファイル中に生成できる命令数を増やす. 命令列の生成法をボトムアップからトップダウンに変更することにより, 命令の種類やバリエーションを増やす. また, クラスフィールドの型や修飾子等の情報を各セクションに挿入することにより, クラスフィールドの型と修飾子がテスト毎に変化するようにする. 本手法に基づくテストシステムを Perl 5 で実装し, Android 9.0 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, 従来のテストシステムに比べてバイトコードの規模とバリエーションが拡大した.

キーワード Android, 仮想マシン, ランダムテスト

Reinforcing Generation of Instruction Sequences in Random Testing of Android Virtual Machine

Ryotaro SHIMIZU† and Nagisa ISHIURA†

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

Abstract This article presents a method of reinforcing generation of instruction sequences in random testing of the Android virtual machine (VM). A mutation based random test generation method, in which input DEX (Dalvik Executable) files are generated by randomly mutating some bytes in existing valid DEX files, ends up with low efficiency because invalid DEX files are rejected by the VM's verifier. On the other hand, in a generation based method, which generates only valid DEX files that pass the verifier, did not incorporate enough number nor enough variation of instructions and class fields. In this article the latter method is extended to provide test files with larger number and variation of instruction and class fields. More than one methods are generated in a byte code to increase the total number of instructions in a file. A top-down generation procedure is employed instead of the bottom-up one to make it easier to afford variety of instructions and operands. The field information in the file header is modified to randomly change the types and modifiers of class fields. A test system has been implemented in Perl 5 and the VM of Android 9.0 have been tested. Although no error has been detected so far, it is expected that the tests provides a wider coverage.

Key words Android, Virtual Machine, Random Tesing

1. はじめに

Android (注1) はモバイル端末向け OS で, スマートフォンやタブレット等に用いられている. さらに最近では, Android は組み込みシステムにも搭載されるなど, 幅広い分野で用いられるようになって来ている. 2017 年には月間ユーザー数が 20

(注1)：https://source.android.com (accessed 2019-01-26)

億人を突破し, 現在もそのユーザー数が増加し続けている (注2). このような利用の拡大に伴い, Android は非常に高い信頼性が求められる.

Android は Linux カーネル, ライブラリ, Android ランタイム, およびアプリケーション等様々なソフトウェアで構成されている. Android の各種ソフトウェアの中で, Android ランタイムは仮想マシンとコアライブラリで構成されている. Android 仮想マシンでは信頼性・性能向上のため頻繁に更新が行われているが, その都度不具合も発生し易く, そのテストは非常に重要な課題である.

仮想マシンのテストは膨大な数のテストケースを用いて徹底的に行われるが, テスト数が有限であるため, これらのテストを経ても不具合が検出できない場合が多く, そのような不具合を検出する方法として, ランダムに生成したプログラムを用いてテストするランダムテストがある.

DexFuzz [1] では, Android 仮想マシンの実行可能ファイルである DEX ファイルをランダムに変異させたものにより, Android 仮想マシンを差分テスト [2] する手法を提案している. 不正な DEX ファイルはベリファイアによって入力時に棄却されてしまうため, ベリファイアの解析によって特定した箇所のみを変異させることにより, DEX ファイルを生成している. また, 3 種の実行パターンで得られた結果を比較し, 計算結果の正誤判定を行っている. しかし, この手法によって生成した全ての DEX ファイルのうち, ベリファイアを通過するのは 10% にすぎず, テストの効率は必ずしも良くない.

文献 [3] では, ランダムな命令列を一から生成することにより, DEX ファイルがベリファイアを 100% 通過するテスト手法を提案している. 命令列をボトムアップに生成することにより, 不正な命令列の生成を防ぎ, 実行結果の照合を可能にしている. しかし, この手法では 1 ファイル中に生成できる命令数および命令種類が少なく, 生成できるクラスフィールド (C 言語の外部変数に相等) の型と修飾子が全てのテストにおいて固定化されていることが課題である.

本稿では, 文献 [3] のテストシステムを拡張し, バイトコードの規模とバリエーションを拡大する手法を提案する. 本手法では, 複数のメソッドを生成することにより, 1 ファイル中に生成できる命令数を増やす. 命令列の生成法をボトムアップからトップダウンに変更することにより, 命令の種類やバリエーションを増やす. また, クラスフィールドの型や修飾子等の情報を各セクションに挿入することにより, クラスフィールドの型と修飾子がテスト毎に変化するようにする.

以下, 本稿では 2 章で ART の構造, DEX ファイル, および Android 仮想マシンのランダムテスト手法について述べ, 3 章では複数のメソッド, 命令列, およびクラスフィールド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. 仮想マシンのランダムテスト

2.1 Android 仮想マシン ART の構成

ART [4] は Android プラットフォームで採用されているレジスタベースの仮想マシンである. 従来は Dalvik VM [5] が Android の仮想マシンとして採用されていたが, Android 5.0 以降から ART に置き換えられた. Dalvik VM は, アプリ起動時に中間コードをネイティブコードに変換する JIT (Just-In-Time) コンパイル方式を併用しているが, ART は アプリインストール時に変換する AOT (Ahead-Of-Time) コンパイル方式を併用することにより, 動作速度の向上と省電力化を図っている.

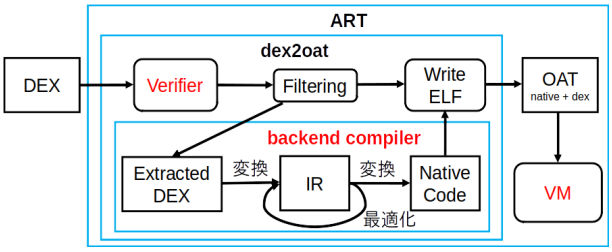

図 1: ART の構成 [6]

ART の構成の概略 [6] を図 1 に示す. DEX (Dalvik Executable) ファイルはバイトコードを含んだ Android 仮想マシンの実行可能形式である. まず, dex2oat のベリファイア (Verifier) が 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だけでなく, 命令のオペランドや分岐命令のジャンプ先等の検査を静的に行う. これには, オペランドのレジスタ番号が命令で使用可能な範囲内であるか, 分岐命令のジャンプ先が命令列の範囲内かつ命令の先頭を指しているか等のチェックも含まれる. DEX ファイルに不正がなければ, フィルタリング (Filtering) で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. フィルタリングは 1 つのメソッド中の命令数が一定の条件を満たしたものだけを抽出する. 即ち, 1 つのメソッド中の命令列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は行われない. フィルタリングにより抽出したバイトコードをバックエンドコンパイラ (backend compiler) で中間表現 (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. そして, ネイティブコードと元の DEX ファイルの内容を組み合わせた ELF 形式の実行可能ファイル OAT を生成し, それを仮想マシンで実行する.

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, および仮想マシンである.

2.2 DEX ファイルの命令列

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 ビットレジスタを持つことができる. 命令の形式はオペランド数によって異なり, 例えば 1 オペランド命令 (`move-result v3`) や 2 オペランド命令 (`add-int/2addr v3, v5`), 3 オペランド命令 (`add-int v3, v5, v6`) 等が存在する. また, オペランドに指定できるレジスタ番号の範囲は 0~15, 0~255, または 0~65535 のいずれかであり, 命令により異なる.

int 型等の 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) に格納される. レジスタペアをオペランドに指定する際, レジスタペアの小さい方のレジスタ番号をオペランドに指定する. レジスタペアのうち, いずれかのレジスタが 32 ビットの値を操作する命令のオペランドに指定されると, フォーマット違反になり, DEX ファイルがベリファイアを通過できなくなる.

命令のオペランドには各セクションのインデックスが指定されることがある. 例えば, メソッドを呼び出す命令

(注2) : https://techcrunch.com/2017/05/17/google-has-2-billion-users-on-android-500m-on-google-photos/ (accessed 2019-01-26)

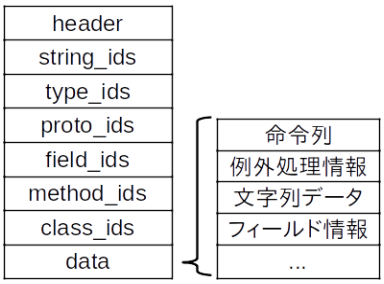

図 2: DEX ファイルの構造

(invoke-static) やクラスフィールドの値を読み書きする命令 (sput, sget) では, それぞれのセクションのインデックスをオペランドに指定する. ただし, セクションの要素数を超えたインデックスを指定することはできない. また, 分岐命令ではジャンプ先の命令が位置するオフセットまでのバイトコードの長さをオペランドに指定する. 分岐命令のジャンプ先が命令の先頭を指していない場合, 不正なジャンプとみなされ, フォーマット違反となる.

演算の結果が未定義になる命令は存在せず, ゼロ除算や配列外参照等の命令を実行すると, 例外が発生する.

2.3 DEX ファイルの構造

DEX ファイルは命令列の他, オフセット情報, 文字列データ, フィールドやメソッドの情報等を持つ. DEX ファイルのフォーマットを図 2 に示す. DEX ファイルには Header, String_ids, Type_ids, Proto_ids, Field_ids, Method_ids, Class_defs, および Data の 8 つのセクションが存在する. Header セクションにはチェックサムや各セクションのサイズ, 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. Data セクションには命令列や例外処理情報, 文字列データ, フィールド情報等が格納されている. 例外処理情報には例外を捕捉するオフセット範囲の情報や捕捉後の処理を行う例外ハンドラ等が格納されており, 文字列データには変数名や型名等 DEX ファイル内で使用される文字列が格納されている. フィールド情報にはフィールドの修飾子を示す値がフィールド毎に 1 つにまとめられて格納されている. String_ids, Type_ids, Proto_ids, Field_ids, Method_ids, および Class_defs セクションにはそれぞれのデータのオフセットやインデックス等が格納されている. String_ids セクションは Data セクションの文字列データに格納された文字列が位置するオフセットを保持している. Field_ids セクションは文字列データに格納された型とフィールド名の文字列を指すインデックスを保持しており, それらのインデックスがフィールド毎に 1 つにまとめられて格納されている.

2.4 ART のランダムテスト

ART のランダムテストに関しては [1] [3] がある.

文献 [1] では, 既存の DEX ファイルをランダムに変異させたものにより ART を差分テストする手法を提案している. ベリファイアの通過率を向上させるため, 既存の DEX ファイルを解析し, オペランドのレジスタ番号やフィールド ID のインデックス等を特定することにより, DEX ファイルを変異させている. また, DEX ファイルを 3 種のオプション毎に実行して得られた結果を比較し, 正誤判定を行っている. この手法により, 189 件のクラッシュを検出しているが, 生成した全ての DEX ファイルのうちベリファイアを通過する DEX ファイルは約 10% であり, テストの効率は必ずしも良くない.

これに対し, 文献 [3] では, ベリファイアを 100% 通過する DEX ファイルを生成する手法を提案している. この手法では既存の DEX ファイルを変異させるのではなく, ランダムな命令列を生成することにより ART のテストを行っている. ゼロ除算等を引き起こす不正な命令列の生成を防ぎ, さらに実行結果が正しいかどうかの照合を可能にするため, 命令列をボトムアップに (後方から前方へ逆順に) 生成する手法を用いている. テストの結果, 生成された DEX ファイルはベリファイアを 100 % 通過することができたが, ART の不具合は検出できていない. その理由としては, 1) ランダムな命令列を生成する箇所が 1 個のメソッド内のみであるため, 命令数が少ない, 2) 3 オペランド以外の演算命令や分岐命令等の命令が生成されていない, 3) クラスフィールドが 20 個しか定義されておらず, その型や修飾子が全てのテストにおいて固定化さていること等が挙げられる.

3. ART のランダムテストにおける命令列生成の強化

3.1 概　　要

本稿では, 文献 [3] の手法を拡張し, バイトコードの規模とバリエーションを拡大することにより, ART のランダムテストを強化する手法を提案する. 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, および仮想マシンである.

dex2oat は 1 つのメソッド中の命令列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. このため 文献 [3] では 1 ファイル中に生成する命令数を制限していたが, 本手法では複数のメソッドを生成することにより, 1 ファイルにより多くの命令を生成できるようにする. また, 命令列のランダム生成法をボトムアップからトップダウンにすることにより, 生成する命令の種類やオペランドのバリエーションを増やす. さらに本手法では DEX ファイルの各セクションに型や修飾子等の情報をランダムに挿入することにより, 型と修飾子がテスト毎に変化するクラスフィールドを生成する.

3.2 複数メソッドの生成

従来法では, クラスフィールドを初期化するメソッド, main メソッドを呼び出すメソッド, およびランダムな命令列を含む main メソッドの 3 メソッドを生成していたが, 本手法ではこれ以外に複数のメソッドを生成して main メソッドから呼び出せるようにする.

バイトコード中に新たなメソッドを追加した場合, DEX ファイルの各セクションにメソッドの型や修飾子等の情報を挿入する必要がある. しかし, これらの挿入によって, 他のデータのオフセットや各セクションにあるインデックス等が変化してしまうため, 整合性を保って修正を行うのは非常に繁雑である. そこで本手法では, 文献 [3] と同様, 空のメソッドを複数定義した Java プログラムをコンパイルして得られる DEX ファイルをテンプレートとし, そのバイトコードのメソッド中にランダムな命令列を生成する方式により, 生成の実装を容易化する.

ただし, テンプレートは Java プログラムから生成するため, メソッド数, 型, 修飾子, 引数の数, および型は全てのテストで固定される.

3.3 命令列の生成

文献 [3] のボトムアップな命令列生成手法では, 命令のオペランドの値を把握して不正な演算を実行する命令の生成を防ぐことができる. しかし ART では, ゼロ除算を除けば不正な演算や結果が未定義になる演算は生じない. そこで本手法では, ラ

```
[000240] seed.<clinit>:()V

// (1) 変数定義ブロック
0x0000: const-wide v0, 3872
0x0005: sput-wide v0, field@0020
...
0x3824: const v0, 2983
0x3827: sput v0, field@2309
0x3829: return-void
```

(a) フィールド定義メソッド

```
[00029c] seed.<init>:()V
0x0000: invoke-direct {v1}, method@0001
0x0003: return-void
```

(b) main 呼び出しメソッド

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

// (1) 変数定義ブロック
0x0000: const v0, 23494
0x0002: const-wide v1, 123
...
0x1131: const-wide v65523, 2142
0x1136: const v65530, 842

// (2) ランダム生成ブロック
0x1139: add-int v12, v234, v75
0x113b: div-int/2addr v9, v15
0x113c: sput v21, field@4341
0x113e: invoke-static {v0, v4, v5, v8, v9}, method@0007
0x1141: move-result v20
...
0x4251: rem-long v231, v167, v55
0x4253: if-eq v14, v8, 12
0x4255: div-long v174, v223, v87
0x4257: move-wide/16 v14655, v10
0x425a: rem-int/lit8 v223, v91, 90
0x425c: return-void

// (3) 例外処理ブロック
0x425e: crashes 32
0x4260:   0x113b - 0x113c
0x4262:   Ljava/lang/ArithmeticException; -> 0x113c
...
0x7601:   0x4251 - 0x4253
0x7603:   Ljava/lang/ArithmeticException; -> 0x4253
0x7605:   0x4255 - 0x4257
0x7607:   Ljava/lang/ArithmeticException; -> 0x4257
0x7609:   0x425a - 0x425c
0x760b:   Ljava/lang/ArithmeticException; -> 0x425c
```

(c) ランダムな命令列

図 3: 生成する DEX のアセンブリコード例

ンダムな命令生成をトップダウンに行う．これにより, 命令列生成の実装が容易になり, 3 オペランド以外の演算命令や分岐命令の生成が可能になる．

本手法で生成する DEX ファイルを逆アセンブルしたアセンブリコード例を図 3 に示す．図 3 では, (1) 変数定義ブロック ((a) の オフセット `0x0000`∼`0x3829`, (c) の オフセット `0x0000`∼`0x1136`), (2) ランダム生成ブロック ((c) の オフセット `0x1139`∼`0x425c`), (3) 例外処理ブロック ((c) の オフセット `0x425e`∼`0x760b`) より構成されている．変数定義ブロックにおいて, (a) の オフセット `0x0000`∼`0x3829` ではクラスフィールドの初期化, (c) の オフセット `0x0000`∼`0x1136` ではレジスタの初期化を行っている．

ランダムな命令列の生成は以下の手順で行う．

1. ランダム生成ブロックの生成

下記の手順を指定された命令数だけ繰り返す．

(i) 生成する命令のオペコードをランダムに 1 つ決定する．

(ii) (i) で選択した命令に応じて, オペランド (レジスタ, クラスフィールド, 定数等) をランダムに決定する．

例えば, 図 3 (c) のオフセット `0x1139` の命令生成では, まず, (i) で int 型の加算を行う命令 `add-int` 命令を選択し, (ii) で `v0`∼`v255` のうち, int 型の値が格納されているレジスタを 3 個選択して, 命令 `add-int v12, v234, v75` を生成する．DEX ファイルでは不正な型のオペランドやレジスタペアの一方のみをオペランドとして扱う命令はフォーマット違反となる．例えば, int 型の値が格納されたレジスタを long 型演算命令 `mul-long` のオペランドとしたり, long 型の値が格納されたレジスタペアの一方を int 型演算命令 `add-int` のオペランドとしてすることは許されない．そこで, 本手法では命令列の生成前に, 各レジスタおよびクラスフィールドに対して型情報を記述することにより, 不正な型のオペランドを選択しないように制御する．

表 1: 本手法で生成する命令

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,`
	`add-long, sub-long, mul-long, div-long,`
	`rem-long, and-long, or-long, xor-long,`
	`shl-long, shr-long, ushr-long,`
	`add-int/2addr, sub-int/2addr,`
	`mul-int/2addr, div-int/2addr,`
	`rem-int/2addr, and-int/2addr,`
	`or-int/2addr, xor-int/2addr,`
	`shl-int/2addr, shr-int/2addr,`
	`ushr-int/2addr, add-long/2addr,`
	`sub-long/2addr, mul-long/2addr,`
	`div-long/2addr, rem-long/2addr,`
	`and-long/2addr, or-long/2addr,`
	`xor-long/2addr, shl-long/2addr,`
	`shr-long/2addr, ushr-long/2addr,`
	`add-int/lit16, sub-int/lit16, mul-int/lit16,`
	`div-int/lit16, rem-int/lit16, and-int/lit16,`
	`or-int/lit16, xor-int/lit16,`
	`add-int/lit8, sub-int/lit8, mul-int/lit8,`
	`div-int/lit8, rem-int/lit8, and-int/lit8,`
	`or-int/lit8, xor-int/lit8,`
	`neg-int, neg-long, not-int, not-long,`
	`int-to-long, long-to-int, cmp-long`

分岐命令	`if-eq, if-ne, if-lt,`
	`if-ge, if-gt, if-le,`
	`if-eqz, if-nez, if-ltz,`
	`if-gez, if-gtz, if-lez,`
	`goto/16, goto/32`

move 命令	`move, move/from16, move/16,`
	`move-wide, move-wide/from16, move-wide/16,`
	`move-result, move-result-wide`

const 命令	`const/16, const,` `const-wide/16, const-wide/32, const-wide`
sput 命令	`sput, sput-wide`
sget 命令	`sget, sget-wide`
invoke 命令	`invoke-static`
nop 命令	`nop`

本手法で生成する命令の一覧を表 1 に示す．本稿の手法では文献 [3] の手法で生成できなかった演算命令の 2 オペランド命令, 分岐命令, move 命令, レジスタに値を設定する命令 (`const`), クラスフィールドの値を読み書きする命令 (`sput, sget`), メソッド呼び出し命令 (`invoke`), および nop 命令を生成する．

分岐命令でループを形成すると無限ループに陥る可能性があるため, 分岐命令のオペランドは正の定数のみとする．また, 分岐命令のジャンプ先が命令の先頭を指していない場合には不正

なジャンプとみなされるため, 本手法では命令列を生成した後にオペランドを修正する. 分岐命令のオペランドは以下の手順で設定する.
(i) 分岐命令より後方の命令をランダムに 1 つ選択する.
(ii) 選択した命令の先頭までのオフセットを算出し, それを分岐命令のオペランドに設定する.

図 3 (c) の 0x4253 の if-eq 命令では, (i) で 3 命令後の rem-int/lit8 命令を選択し, (ii) で分岐命令の先頭オフセットから rem-int/lit8 命令の先頭オフセットまでのバイトコードの長さ (12) を分岐命令のオペランドに設定する.

メソッド呼び出しに関しては, 呼び出しの深さが増えると実行時間が増えてテストの効率が下がるため, メソッド呼び出しを行う invoke 命令の生成箇所は main メソッド内に限定する. メソッド呼び出し命令の生成は以下の手順で行う.
(i) 引数となるレジスタをランダムに設定する.
(ii) main メソッド以外のメソッドのインデックスの中からランダムに 1 つ選択する.
(iii) 呼び出すメソッドの戻り値をレジスタに格納する命令 (move-result) を生成する.

図 3 (c) の オフセット 0x113e の命令生成では, まず, (i) で, レジスタ v1, v4, v5, v8, v9 を引数として選択し, (ii) で, (c) の メソッドのインデックス method@0007 を選択する. そして, (iii) で int 型の戻り値をレジスタに格納する命令 move-result をオフセット 0x1141 に生成する.

2. 例外処理ブロックの生成

本稿の手法では, 命令のオペランドの値の制御は行っていないため, 除算命令と剰余命令でゼロ除算が発生する可能性がある. そこで本手法ではゼロ除算の例外処理を行うバイトコードの生成を追加する. 除算命令と剰余命令を例外の捕捉範囲として指定し, 例外が発生した場合には, その次の命令にジャンプするように設定する. 例えば, 図 3 (c) の オフセット 0x113b の div-int/2addr 命令でゼロ除算が発生した場合, オフセット 0x113b ∼ 0x113c の範囲を捕捉する情報をオフセット 0x4260 に生成する. そして, 例外を捕捉すると 0x113c へジャンプする処理を行う例外ハンドラをオフセット 0x4262 に生成する. そして, 生成した例外ハンドラの数 (32) をオフセット 0x425e に設定する.

3. 変数定義ブロックの生成

文献 [3] のボトムアップ命令列生成手法では, レジスタに期待値を格納するような命令列を生成し, その命令列に必要になる初期値をレジスタに設定していた. しかし, 本手法ではトップダウンに命令列を生成するため, 初期値をランダムに決定してレジスタとクラスフィールドに設定する命令列を生成する. 初期値については定義されたレジスタとクラスフィールドの型から初期化する値の範囲を決定し, その中からランダムに値を 1 つ選択する. 図 3 (c) の オフセット 0x0000 では, 生成したレジスタ v0 の型が int 型であるため, int 型の値の範囲の中から値 23494 を選択し, その値を v0 にロードする命令 (const) を生成する.

3.4 クラスフィールドの生成

文献 [3] では, Java プログラムにクラスフィールドを定義してコンパイルすることにより, DEX ファイルにクラスフィールドを生成していたため, クラスフィールドの型と修飾子が全てのテストにおいて固定化されていた. 本手法では, クラスフィールドの名前, 型, および修飾子の情報を DEX ファイルの各セクションに挿入することにより, DEX ファイルにクラスフィールドを生成する. これにより, 型と修飾子がテスト毎に変化するようなクラスフィールドの生成が可能である.

表 2: クラスフィールドの修飾子

	値	意味
final	0x0010	値の上書きを禁止する
volatile	0x0040	コンパイラの最適化を抑制する
transient	0x0080	シリアライズの対象から除外する
synthetic	0x1000	コンパイラが自動的に生成する
enum	0x4000	複数の変数を 1 つにまとめる

Field_ids セクションにはフィールドの型と名前の情報がフィールド毎に 1 つにまとめられて格納されているため, Field_ids セクションにクラスフィールドの型と名前の情報を挿入する.

ただし, これらの情報は Data セクションの文字列データに格納された文字列を指すインデックスであるため, 文字列データに型とクラスフィールド名の文字列を保持しなければならない. 文字列データには型の文字列が既に生成されているが, クラスフィールド名の文字列が生成されていないため, 文字列データにクラスフィールド名の文字列を挿入する.

また, Data セクションのフィールド情報にはフィールドの修飾子情報が格納されているため, フィールド情報にクラスフィールドの修飾子情報を挿入する. クラスフィールドの修飾子情報は表 2 に示された値であり, その中からランダムに 1 つ選択して決定する. ただし, クラスフィールドは静的な変数であり, static 修飾子が必要であるため, 全てのクラスフィールドの修飾子情報に static を示す値 0x0008 を加算する.

String_ids セクションは Data セクションの文字列データに格納された各文字列が位置するオフセットを保持しており, 本手法では文字列データにクラスフィールド名の文字列を挿入したため, それらの文字列が位置するオフセットを String_ids セクションに挿入する.

4. 実験結果

提案手法に基づくテストシステムを Perl 5 で実装した. 本システムは Linux (Ubuntu) で動作する.

本手法で生成した DEX ファイルにより, ART をランダムテストする実験を行った. 生成した DEX ファイルは, adb (Android Debug Bridge) ツールを用いて Android のエミュレータに転送して実行した. 実験は Ubuntu 14.04 LTS で行い, ART のバージョンは 2.1.0, Android のエミュレータのバージョンは Android 9.0 とする. また, dex2oat の最適化オプションは “Optimizing” とした. この実験において生成する DEX ファイルに定義したメソッド数は 6 個 (int 型 3 個, long 型 3 個), クラスフィールド数は 65536 個とし, メソッド毎におけるランダムに生成する命令数は 1000 個とした.

実験の結果を表 3 に示す. “時間” はテストに要した時間, “#test” はテストプログラム数, “#error” 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. テストの結果, 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.

本手法と文献 [3] で生成したテストのカバレッジの比較を表 4 に示す. 表中の数値は, 100 個の DEX ファイルの平均値である. 本手法ではメソッドを複数生成しているため, 文献 [3] の手法に比べて 1 ファイル中にランダムに生成される命令数が増加している. 生成速度が従来法に比べておよそ 4 分の 1 にまで減

表 3: 実験結果

time [h]	#test	#error				
		verify	quicken	space	speed	everything
320	45,714	0	0	0	0	0

(Ubuntu 14.04 LTS, Xeon 3.60GHz, RAM: 16GB)

表 4: 生成したテストのカバレッジの比較

		文献 [3]	本手法
メソッド数		3.0	9.0
命令数		1000.0	7000.0
生成速度 [個/h]		756.0	180.0
分岐命令数		0.0	1064.4
メソッド呼び出し命令数		0.0	12.2
例外処理数		0.0	853.6
命令種類		27.0	100.0
レジスタアクセス数 (演算命令)		3.0	250.1
フィールドアクセス数		20.0	288.3
レジスタ番号 (演算命令)	最小値	10.0	2.0
	最大値	14.0	255.0
フィールド ID	最小値	1.0	381.8
	最大値	21.0	65094.7
フィールド数		20.0	65536.0
フィールド修飾子の種類		3.0	12.0
ファイルサイズ [kB]		8.5	1562.9

少したが, 生成したランダムな命令数については 7 倍にまで増加したため, 命令列の生成効率は従来法より向上していると言える. 本手法では, 従来法で生成できなかった分岐命令, メソッド呼び出し命令, および例外処理を含んだ命令列を生成しているため, 従来法に比べて本手法で生成した命令種類が増加している. また, 従来法に比べて演算命令でアクセスされたレジスタ数とクラスフィールド数も増加し, より広範囲にアクセスできている. 本手法では DEX ファイルの各セクションにクラスフィールドの型や修飾子などの情報を挿入したことにより, 従来法に比べて多くのクラスフィールドを生成することができ, それらの修飾子の種類も増加している.

エラーを検出できなかった原因としては, バイトコードの規模が依然として小さいことが考えられる. 本手法では, ランダムな命令列を処理する複数のメソッドを生成することができたが, まだ 1 つのクラス内でしか命令列を生成していない. これを解決するためには, 複数のクラスを生成し, インスタンスを組み合わせてより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, 生成するオペランドの型が int 型と long 型に限定されているため, 配列やクラス等オブジェクトを格納したオペランドも生成することが必要と考えられる.

5. むすび

本稿では, Android 仮想マシンのランダムテストにおける命令列生成の強化を行う手法を提案した. 実験の結果, Android 9.0 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, 従来法のテストシステムに比べて, バイトコードの規模とバリエーションを拡大することができた.

今後の課題としては, 複数のクラスの生成, オブジェクトを用いた命令の生成等によるバイトコード生成の強化が挙げられる.

謝辞

本研究に関してご協力, ご討議頂いた池尾弘史氏 (現 NTT コミュニケーションズ株式会社) をはじめ, 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] 池尾弘史, 清水遼太朗, 石浦菜岐佐: “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,” 信学技報, VLD2017-95 (Feb. 2018).

[4] Android Open Source Project (Android core technologies), `https://source.android.com/devices/tech/dalvik/` (accessed 2019-01-26).

[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).