Vol.2014-CSEC-66 No.7
Vol.2014-SPT-10 No.7
2014/7/3

Androidにおける高速な簡易スタックトレースの実現とパーミッション制御手法への応用

高瀬 拓歩[1,a)]　日置 将太[1]　齋藤 彰一[1]　瀧本 栄二[2]　毛利 公一[2]　松尾 啓志[1]

概要：アプリケーションの開発では，第三者によって開発されたライブラリが利用されることが一般的である．しかし，導入したライブラリによって情報漏洩が引き起こされる事例が発生している．特に Android では，アプリケーションに組み込んだ広告ライブラリが，個人情報を悪用することが問題となっている．これに対して，実行元クラスをアプリケーションとライブラリで区別することで動作を制限する研究が行われている．しかし，その区別に用いるスタックトレースのオーバヘッドが大きく，アプリケーションの実行速度を低下させる．本稿では，Android において実行元クラスを判断するために必要な情報のみを取得できる簡易スタックトレースを実現し，これをパーミッション制御機構に適用した．これにより，既存の制御機構より性能低下を抑えた実行元クラス判断手法を実現した．

キーワード：Android 権限分離 スタックトレース 第三者ライブラリ 実行元クラス判断

Implementation of Simple StackTrace and its Application to Permission Control System in Android

Takuho Takase[1,a)]　Shota Hioki[1]　Shoichi Saito[1]　Eiji Takimoto[2]　Koichi Mouri[2]
Hiroshi Matsuo[1]

1. はじめに

アプリケーションの開発では，Application Programming Interface（以下，API という）を用いて OS が提供する機能を利用することに加えて，第三者の開発したライブラリ（以下，第三者ライブラリという）を自らのアプリケーション（以下，ホストアプリクラスという）に組み込んで利用できる．しかし，第三者ライブラリを利用することでシステムに障害が発生したり，情報漏洩が引き起こされる可能性がある．第三者ライブラリは様々な機能をもったものが存在する．Android[1] のビジネスモデルはアプリケーションを無料で配布しアプリケーションに組み込んだ広告で収入を得る形が一般的であるため，AdMob[2] に代表される広告ライブラリは多くのアプリケーションに組み込まれている．しかし，文献 [3] によると，ユーザに無断でユーザの情報を収集する広告ライブラリが存在することが報告されている．また，文献 [4] によると，広告ライブラリにセキュリティリスクが存在することが報告されている．これは，携帯電話端末には，電話番号，氏名，Web 閲覧履歴，連絡帳などの情報が存在し，攻撃者にとっては有益な情報が多いためである．このように，アプリケーションの利用者はアプリケーション本体だけでなく，ライブラリに対しても注意を払う必要がある．

Android ではアプリケーションに対するセキュリティの仕組みとしてサンドボックスとパーミッション機構がある．サンドボックスとはアプリケーションを保護された領域で実行する機構である．これによって，アプリケーションが Android のシステムや他のアプリケーションへアクセスすることを防止している．パーミッション機構は端末の一部の情報や機能の利用を制限する仕組みである．アプリケー

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 立命館大学
Ritsumeikan University
[a)] takase@matlab.nitech.ac.jp

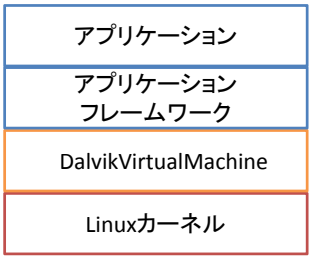

図 1　Android のアーキテクチャ

ションがパーミッション機構の対象となる端末の情報や機能を利用するには，アプリケーションのインストール時にユーザの承認が必要である．また，パーミッションは，アプリケーション単位で管理される．しかし，先ほど述べたように多くのアプリケーションには第三者ライブラリが含まれるが，ホストアプリクラスと第三者ライブラリに対して許可されるパーミッションは区別されない．このため，ホストアプリクラスが要求したパーミッションを第三者ライブラリが利用して，情報を漏洩する問題が発生している．第三者ライブラリの一種である広告ライブラリでも，これを悪用して情報の収集が行われる事例が報告されている．

パーミッションが必要な API を実行した際にその API を実行したクラスが，ホストアプリクラスか第三者ライブラリかを区別する研究として細粒度アクセス制御機構 [5] やパーミッション制御機構 [6][7] がある．細粒度アクセス制御機構では，API が実行された際に実行元クラスを特定して，ユーザに実行元クラスを通知した上で許可を求める．また，パーミッション制御機構では，あらかじめパーミッションをホストアプリクラスに与えられたものと，第三者ライブラリに与えられたもので別々に管理している．これらの研究では実行元クラスを特定する際にスタックトレースを用いている．しかし，スタックトレースを取得するためのコストがオーバヘッドとなる．これによりアプリケーションの実行速度の低下やレスポンスの遅延の発生が考えられる．そこで，本稿では実行元クラスを特定するために必要な情報のみを取得できる高速なスタックトレースを実現し，これをパーミッション制御機構に適用した．これにより，既存の制御機構より性能低下を抑えた実行元クラス判断手法を実現した．

以下，2 章で Android のアーキテクチャについて簡単に述べる．3 章で関連研究，4 章で提案方式，5 章でその実装について述べ，6 章で評価と考察を述べる．最後に 7 章でまとめる．

2. Android のアーキテクチャ

Android のアーキテクチャの概要を図 1 に示す．Android は，Linux カーネル，Dalvik Virtual Machine（以下，DalvikVM という），アプリケーションフレームワーク，アプリケーションから構成されている．Linux カーネルは AndroidOS のカーネルであり，セキュリティ，メモリ管理，プロセス管理，ネットワークスタック，ドライバモデルなどのコアとなるシステムサービスを提供する．Linux カーネルの上で DalvikVM が動作する．DalvikVM は，JavaVM をベースに低メモリ環境向けに最適化された VM である．さらに，複数の DalvikVM のインスタンスを同時に動作させることが可能で，Android アプリケーションはすべてこの DalvikVM 上で動作する．アプリケーションフレームワークはアプリケーションに API を提供すると共に，アプリケーションの管理を行っている．アプリケーションフレームワークでは，AcvitivityManager や ContentsProviders などの管理アプリケーションが動作している．アプリケーションはプリインストールされているブラウザや電話などの他に，ユーザがインストールしたアプリケーションなどが存在する．本稿では，アプリケーションフレームワーク及び，DalvikVM に改良を加えることで提案の実現を行う．

3. 関連研究

本章では，権限分離に関する研究について述べる．3.1 節で Android のパーミッション機構について述べ，3.2 節で権限分離の研究であるパーミッション制御機構と細粒度アクセス制御機構の詳細を述べる．そして 3.3 節でそれらの問題点について述べる．

3.1 パーミッション機構

本節では Android のパーミッション機構についてその概要と動作を述べる．

3.1.1 概要

Android のすべてのアプリケーションは，異なるプロセスとして異なるサンドボックスの中で実行される．そのため，アプリケーションは Android のシステムや他のアプリケーションの情報へアクセスすることができない．サンドボックス内のアプリケーションが，保護された情報へアクセスするための機構としてパーミッション機構がある．パーミッションをユーザが許可することによって，保護された情報を取得することができる．パーミッションが必要な情報は，ネットワークアクセス，電話帳情報，電話番号情報，端末識別子情報，SD カードへの書き込み，カメラの使用などが挙げられる．パーミッションが必要な情報をアプリケーションが利用するためには，アプリケーションのインストール時にユーザの承認をとる必要がある．ユーザは，アプリケーションが要求するパーミッションを承認してアプリケーションをインストールするか，パーミッションの承認を行わずアプリケーションをインストールしないかを選択する．アプリケーションの実行時にはパーミッションが承認されているかどうかの検証が行われ，アプリケーションの動作を制御する．アプリケーションがイ

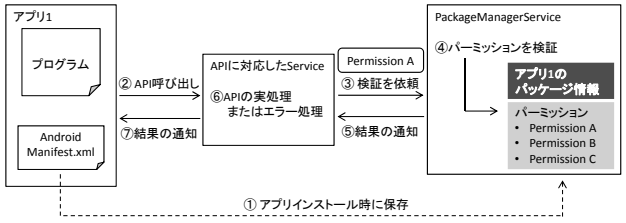

図 2 API 呼び出し時のパーミッション機構の動作

ンストールされた後に，要求するパーミッションを変更することや，ユーザが承認していないパーミッションをアプリケーションが用いることはできない．

3.1.2 動作

パーミッション機構の動作を図 2 に示す．まず，アプリケーションが端末にインストールされるとき，マニフェストファイルに書かれた情報が PackageManagerService 上にパッケージ毎に保存される（図 2 の①）．マニフェストファイルとは，アプリケーション毎に必ず一つ存在する，アプリケーションに関する必要不可欠な情報を Android システムに伝えられるファイルである．アプリケーションが API を呼び出すと，プロセス間通信を用いて対応するサービスに処理が移る（図 2 の②）．当該 API の実行にパーミッションが必要な場合は，対応するサービスは PackageManagerService に対してパーミッションの検証を依頼する（図 2 の③）．このとき，検証対象となるパーミッションの名前を PackageManagerService に渡す．PackageManagerService では，API を実行したアプリケーションのパッケージの情報を参照し，検証対象のパーミッションを含んでいるか否かを検証し（図 2 の④），結果を返す（図 2 の⑤）．検証結果を受け取った API に対応するサービスでは，検証が成功していれば正常に処理を行い，失敗していればセキュリティエラーとして処理を失敗させ（図 2 の⑥），結果を通知する（図 2 の⑦）．

3.2 パーミッション制御機構と細粒度アクセス制御機構

パーミッションの実行元クラスを判別する研究であるパーミッション制御機構と細粒度アクセス制御機構は，パーミッションが必要な API の実行元クラスを，ホストアプリクラスと第三者ライブラリで区別する点は同じである．

パーミッション制御機構では，Android マニフェストファイルの書式を変更し，ホストアプリクラスが要求するパーミッションと第三者ライブラリが要求するパーミッションを別々に記述する．マニフェストファイルに記された情報は，アプリケーションが実行される前に Android の PackageParser に伝えられ，アプリケーションフレームワークの PackageManagerService がその情報を保持する．アプリケーションが API を使用する場合には，PackageManagerService がパーミッションの検証を行う．この時，パーミッション制御機構では，アプリケーションに組み込んだ制御コードがスタックトレースを用いて実行元クラスの情報を API 呼び出しに付与して PackageManagerService に通知することで，実行元クラスに基づいた検証を行う．パーミッション制御機構では，このように，実行元を判断するためにスタックトレースを用いている．

既存の Android では一つのパーミッションに対して複数の機能や情報が含まれるが，細粒度アクセス制御機構では，パーミッションの粒度を API 毎に細かく設定することで，アプリケーションが使用するパーミッションがどのような機能や情報を使用しているのか明確にしている．また，既存の Android では，アプリケーションのインストール時にユーザが承認したパーミッションはそれ以降ユーザへの確認なくアプリケーションは使用することができる．しかし本機構では，API が使用されたときに実行元クラスを判断して，ユーザに実行元クラスと実行する API を提示する．そして，ユーザが承認を行わないと API を実行することができない．実行元クラスを判断するには，パーミッション制御機構と同様にスタックトレースを用いてクラス名を取得し，アプリケーションのパッケージ名と比較することで実行元がホストアプリクラスか第三者ライブラリかを区別する．

3.3 関連研究の問題点

関連研究では，いずれも実行元を特定するためスタックトレースを用いている．両研究では API の呼び出し頻度が高くないことからシステムへ与える影響は低いと述べているが，従来の Android にはなかったオーバヘッドが発生しており，アプリケーションの処理速度に影響を与えている．

4. 提案方式

本章では，まず既存の実行元クラスの判断方法とその動作概要を述べる．次に，スタックトレースのオーバヘッドを軽減させる方法として，実行元クラスを記憶して高速化する方法とスタックトレースを簡易化して高速化する方法の二つについて述べる．最後に二つの方法を比較して提案方式について述べる．

4.1 既存の実行元クラス判断方法

関連研究の実行元クラスを判断する方法は，スタックトレースを用いるものである．Android において，スタックトレースを取得するには，getStackTrace メソッドを使用することが一般的である．DalvikVM に保存されているコールスタックにはメソッド構造体へのポインタとメソッドが終了した際の戻りアドレスの組が記憶されている（図 3 の①）．getStackTrace メソッドが実行されると，コールスタックから StackTraceElement の配列を生成する（図 3 の②）．StackTraceElement にはクラス名，メソッド名，ファイル名，実行したメソッドの行番号の情報が含まれる．3

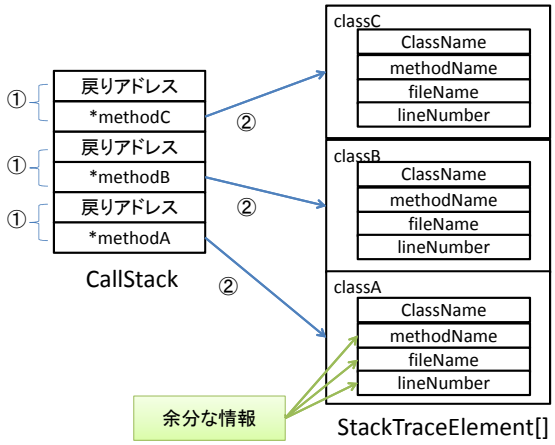

図 3　既存のスタックトレース

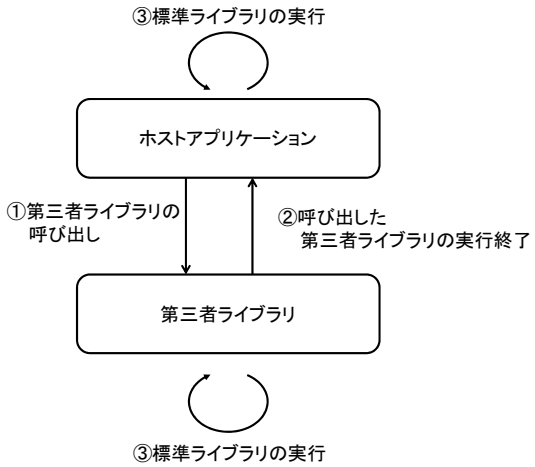

図 4　実行元クラスの記憶

章で述べた関連研究での実行元クラスの判断においては，これらの要素の内のクラス名のみを使用し，事前に定義されたホストアプリクラスに属するクラスかそれ以外のクラスかを判断する．

4.2　方法 1:実行元クラスの記憶

本節では，初めに実行元クラスを記憶する方法の概要を述べ，次にその動作を述べる．

4.2.1　概要

関連研究では，実行元クラスをスタックトレースを用いて特定するためのコストがオーバヘッドとなっている．そこで，あらかじめ実行元クラスを DalvikVM で記憶する方式について検討する．実行元クラスの記憶とは，アプリケーションが現在ホストアプリクラスのコードを実行中なのか，第三者ライブラリのコードを実行中なのか，DalvikVM が判断して記憶する方法である．これにより，実行元クラスを特定する処理が，DalvikVM で記憶しておいた判定結果を返すだけとなり，関連研究が抱える問題点であるオーバヘッドを抑制できる．

4.2.2　動作

一般にアプリケーションは，ホストアプリクラスと第三者ライブラリから構成される．アプリケーションの実行中には，それらとは別に，print メソッドなど Android 標準ライブラリのメソッドを実行する．しかし，Android 標準ライブラリが主体的に悪意ある動作をすることはないため，実行元クラスとしての記憶は，ホストアプリクラスか第三者ライブラリのどちらかである．DalvikVM で記憶する実行元クラスの遷移を図 4 に示す．アプリケーションでは先に必ずホストアプリクラスが実行されるので，初めの実行元クラスはホストアプリクラスである．次にホストアプリクラスから第三者ライブラリのメソッドが呼ばれたとき，実行元クラスは第三者ライブラリへと遷移する（図 4 の①）．第三者ライブラリのメソッドが終了したとき実行元クラスはホストアプリクラスへと遷移する（図 4 の②）．標準ライブラリのメソッドが実行されたときは実行元クラスは元のまま遷移しない（図 4 の③）．

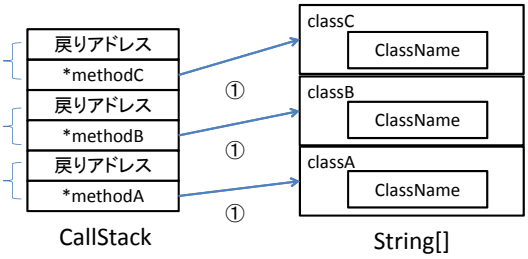

図 5　簡易スタックトレース

4.3　方法 2:スタックトレースの簡易化

getStackTrace メソッドを用いた場合，StackTraceElement の配列が生成される．ここで，StackTraceElement の要素のうちクラス名以外の要素，つまりメソッド名，ファイル名，行番号といった要素は実行元クラスの判断には余分な情報となる．そこで，クラス名のみの配列を生成する簡易スタックトレースを作成する方式を検討する．簡易スタックトレースの動作概要を図 5 に示す．簡易スタックトレースでは，コールスタックから生成するインスタンスを，実行元クラス特定に必要なクラス名のみと簡易化する（図 5 の①）．これにより，DalvikVM における StackTraceElement を構成する多様なクラスインスタンスの生成時間や利用メモリ量の減少が期待でき，全体として高速にメソッドを実行することが可能となる．その結果，実行元クラスの特定に要するオーバヘッドも減少可能である．

4.4　比較

二つの方法はそれぞれ利点と欠点を含んでいる．実行元クラスを記憶する方法では，実行元クラスの特定の際の処理は，DalvikVM で記憶している判定結果を返すのみのため，スタックトレースの簡易化方式より高速に動作すると考えられる．しかし，実行元クラス記憶のための処理が，新たなオーバヘッドとなると考えられる．また，実行元ク

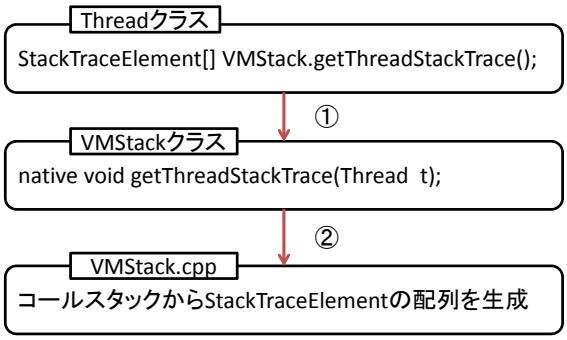

図 6　スタックトレースの流れ

ラスを DalvikVM で判断するため，その判定結果の保護も行う必要がある．一方，スタックトレースの簡易化の方法では，実行元クラスを記憶する方式と比較して，実行元クラスの特定時にコールスタックを探索する必要があるため，特定処理は遅いと考えられる．しかし，既存のスタックトレースと比べて高速に動作することは明らかで，また，新たなオーバヘッドは発生しない．

また，実行元クラスの記憶方法では，パーミッション機構以外の場所で新たにセキュリティ機構を構築することとなる．このために，判断の元となるクラス情報を DalvikVM にも保存することとなり，セキュリティ情報の分散による管理コストの増加が問題となる．そこで，本稿ではスタックトレースの簡易化を提案する．

5. 実装

提案方式の簡易スタックトレースを PandaBoard ES[8] Android4.2 に実装した．また，簡易スタックトレースをパーミッション制御機構に適用した．本章では既存と提案のスタックトレースの実装の詳細について述べる．

5.1 既存のスタックトレースの実装

既存のスタックトレースの実装について述べる．既存のスタックトレースは，Java の API である getStackTrace メソッドを用いるものである．getStackTrace メソッドは Thread クラスのメソッドであり，StackTraceElement の配列を返す．getStackTrace メソッドの内部では VMStack クラスの getThreadStackTrace メソッドを呼び出す（図 6 の①）．getThreadStackTrace メソッドは Thread インスタンスを引数に StackTraceElement の配列を返す．getThreadStackTrace メソッドは Java Native Interface（以下，JNI という）で実装されたメソッドである（図 6 の②）．JNI で実装された getThreadStackTrace メソッドは DalvikVM 内部のコールスタックから StackTraceElement の配列を生成する．次に，DalvikVM 内部でのスタックトレース生成の擬似コードを図 7 に示す．図 7 の各部の処理を以下に示す．

① DalvikVM が記憶しているスレッドリストから，アプリケーションが動いていたスレッドを特定する．

② 特定したスレッドのコールスタックの深さを求める．

③ コールスタックから，クラス名，メソッド名，ファイル名，行番号を取得し，StackTraceElement の配列を生成する．

④ StackTraceElement のポインタをリターンする．

```
StackTraceElement[] getThreadStackTrace(Thread thread){
    while(1){   /* アプリケーションのスレッドを特定 */
①       if(thead == threadlist[threadIndex]) break;
        threadIndex++;
    }
    stackdepth = getstackdepth(threadlist[threadIndex]);
②          /* スタックの深さを求める */

    for(stackIndex=0 ; stackIndex < stackdepth ; stackIndex++){
            /* コールスタックからStackTraceElementの配列を生成 */
③       stacktraceelement[stackIndex].className = CallStack[stackIndex].cname;
        stacktraceelement[stackIndex].methodName = CallStack[stackIndex].methname;
        stacktraceelement[stackIndex].fileName = CallStack[stackIndex].fileName;
        stacktraceelement[stackIndex].linenumber = CallStack[stackIndex].linenumber;
    }
④ return (stacktraceelement);
}
```

図 7　DalvikVM 内部でのスタックトレースの擬似コード

```
String[]  NewgetThreadStackTrace(Thread thread){
    while(1){   /* アプリケーションのスレッドを特定 */
        if(thead == threadlist[threadIndex]) break;
        threadIndex++;
    }
    stackdepth = getstackdepth(threadlist[threadIndex]);
            /* スタックの深さを求める */

    for(stackIndex=0;stackIndex<stackdepth;stackIndex++){
①           /* コールスタックからString配列を生成 */
        className[stackIndex] = CallStack[stackIndex].cname;
    }

    return (className);
}
```

図 8　簡易スタックトレースの擬似コード

5.2 簡易スタックトレース

簡易スタックトレース作成のために，5.1 節で述べた既存の 3 つのメソッドに対応する新メソッドを追加した．Thread クラスと VMStack クラスには，簡易スタックトレース用の JNI メソッドを呼び出すためのメソッドを追加した．また，DalvikVM には，JNI を用いて簡易スタックトレースを生成するためのメソッドを作成した．簡易スタックトレースの擬似コードを図 8 に示す．既存のスタックトレースと比べて図 8 の①に示すようにクラス名の特定だけを行うことで処理の削減を行った．

5.3 パーミッション制御手法への適用

パーミッション制御手法では既存のスタックトレースを用いて StackTraceElement の配列を生成する．その上で，クラス名のみを使用している．しかし，簡易スタックトレースを使うことでその処理も省略が可能である．提案方式は，既存のパーミッション制御手法と同様に，アプリケーションのソースコードに変更は必要がない．

表 1　評価環境

測定端末	PandaBoard ES
CPU	1.2GHz ARM Cortex-A9 デュアルコア
memory	1GB
OS	Android OS 4.2 に機能を追加

```
Text
FATAL EXCEPTION: main
java.lang.RuntimeException: Unable to start activity Component
Info{my.com.example.permissiontest/my.com.example.permissionte
st.MainActivity}: java.lang.SecurityException: Requires READ_P
HONE_STATE: Neither user 10041 nor current process has android
.permission.READ_PHONE_STATE.
```

図 9　第三者ライブラリによるパーミッションエラーのログ

6. 評価

本章では，初めに，提案手法をパーミッション制御手法に組み込んだ際にシステムが正しく動作することを検証する．次に，提案手法の実行時間を評価する．Android ではアプリケーションの実行中にメモリ領域が足りなくなると自動で GC が実行されるが，GC の動作には非常に時間がかかり，GC が発生する場合と発生しない場合で実行時間に差が生じるため，それぞれの場合で別々に評価を行う．また，実行時間の取得には Java の nanoTime メソッドを用いた．評価には表 1 に示すスペックを持つ PandaBoard ES を用いた．

6.1 動作検証

ホストアプリクラスと第三者ライブラリで構成する評価用アプリを作成して動作検証を行った．この評価用アプリは，ホストアプリクラスと第三者ライブラリの双方が，パーミッションの READ_PHONE_STATE を必要とする getLine1Number メソッドを実行する．パーミッションとして，ホストアプリクラスには READ_PHONE_STATE を与え，第三者ライブラリのクラスには同パーミッションを与えない．Android SDK 付属の logcat を用いて取得した実行時のログを図 9 に示す．これは，第三者ライブラリが getLine1Number メソッドを実行しようとしたときに，READ_PHONE_STATE のパーミッションがないためエラーが発生しメソッドを実行できなかったことを示している．このように，パーミッション制御機構と同様にホストアプリクラスと第三者ライブラリでパーミッションの制御を正しく行うことを確認した．

6.2 GC が発生しない場合

初めに，提案手法の簡易スタックトレースと既存のスタックトレースの実行時間について比較する．それぞれの処理を GC が発生しない回数である 50 回ずつ実行し，その合計時間を 10 回計測してその平均値をとった．スタックの深さが 17 の場合の結果を表 2（50 回）に示す．提案手法を用いた簡易スタックトレースは 6.02 ミリ秒（一回あたり 120 マイクロ秒）と既存のスタックトレース処理の 18.2 ミリ秒（一回あたり 364 マイクロ秒）と比べて約 67%の処理時間を削減した．

表 2　スタックトレースの所要時間

実行回数	50 回（GC 無）（1 回あたり）	5 万回（GC 有）（1 回あたり）
既存スタックトレース	18.2[msec]（364[usec]）	23.6[sec]（472[usec]）
提案スタックトレース	6.02[msec]（120[usec]）	8.80[sec]（176[usec]）

表 3　API の所要時間

実行回数	50 回（GC 無）（1 回あたり）	5 万回（GC 有）（1 回あたり）
標準状態の Android	39.5[msec]（790[usec]）	39.6[sec]（790[usec]）
従来のパーミッション制御機構	68.9[msec]（1380[usec]）	83.1[sec]（1660[usec]）
簡易スタックトレース手法	52.7[msec]（1050[usec]）	61.4[sec]（1230[usec]）

次に，標準の Android，パーミッション制御機構を実装した Android，パーミッション制御機構に提案手法を組み込んだ Android の三つで，パーミッションが必要な API である getLine1Number を実行し，実行時間を比較した．スタックトレースの場合と同様に，API の実行を 50 回実行し，その合計時間を 10 回計測して平均値をとった．結果を表 3（50 回）に示す．標準状態の Android の 39.5 ミリ秒（一回あたり 790 マイクロ秒）と比べて，提案手法では 52.7 ミリ秒（一回あたり 1050 マイクロ秒）と約 33%実行時間が増加したが，パーミッション制御手法の 68.9 ミリ秒（一回あたり 1380 マイクロ秒）と比べて提案手法では，約 24%の処理時間の高速化に成功した．

API を実行したとき，標準状態の Android と比べて提案手法では一回あたり 260 マイクロ秒実行時間が増加している．これは，スタックトレースのオーバヘッドの 120 マイクロ秒を考慮しても，140 マイクロ秒余分である．同様に，標準状態の Android と従来のパーミッション制御手法を比べた場合も 230 マイクロ秒余分にある．これらは，スタックトレース後の処理で増加していると考えられるが，詳細は不明であり，今後原因を特定する必要がある．

6.3 GC が発生する場合

スタックトレースの処理において GC が発生するようにスタックトレース処理を 5 万回行い，その合計時間を 10 回計測して平均値をとった．結果を表 2（5 万回）に示す．提案手法を用いた簡易スタックトレースは 8.80 秒（1 回あたり 176 マイクロ秒）と既存のスタックトレース処理の 23.6 秒（1 回あたり 472 マイクロ秒）と比べて約 63%の処理時間を削減した．

次に API の getLine1Number についても同様に，5 万回

表 4　API を 5 万回実行した場合の GC の発生回数

	発生回数
標準状態の Android	50
従来のパーミッション制御機構	746
簡易スタックトレース手法	366

の処理を行い，その合計時間を 10 回計測して平均値をとった．結果を表 3（5 万回）に示す．標準状態の Android の 39.6 秒（一回あたり 790 マイクロ秒）と比べて提案手法では 61.4 秒（一回あたり 1230 マイクロ秒）と約 55%実行時間が増加したが，パーミッション制御手法の 83.1 秒（一回あたり 1660 マイクロ秒）と比べて提案手法では，約 26%の処理時間の高速化に成功した．

最後に，API の getLine1Number を 5 万回実行した際の GC の発生回数を計測した．結果を表 4 に示す．標準状態の Android と比べて提案手法では，約 7 倍の GC が発生したが，パーミッション制御手法と比べて提案手法では，約 51%の発生回数の削減に成功した．

GC の発生の有無で実行時間の増加量を比べると，スタックトレースの場合は，既存のスタックトレースが一回あたり 108 マイクロ秒の増加に対して，提案のスタックトレースは一回あたり 56 マイクロ秒の増加である．また，API の場合は，従来のパーミッション制御手法が 280 マイクロ秒の増加に対して，提案を組み込んだパーミッション制御手法は 180 マイクロ秒の増加である．いずれの場合も GC の発生によって実行時間が増加しているが，提案手法の方が増加量が少ない．これは，提案手法ではスタックトレースの簡易化によって，使用するメモリ量が減り，GC の発生回数が減少したためである．このことから，GC の発生という観点からも提案手法は有用であるといえる．

7. まとめ

本稿では，スタックトレースの動作がアプリケーションに与える影響を減少させるために，簡易スタックトレースを用いる方式を提案した．既存のスタックトレースが実行元クラスの特定に必要な情報だけでなく，余分な情報も収集するのに対して，提案方式は実行元クラスの特定のために必要な最小限の情報のみを収集する．

提案方式を実装し PandaBoard ES で評価を行った．その結果，スタックトレースの処理時間を，最小で通常実行の約 33%に削減した．パーミッション制御手法に組み込んだ場合には，標準のスタックトレースを利用する場合と比較して最小で約 74%の実行時間に削減した．さらに，メモリ使用量の削減により GC の発生頻度はパーミッション制御手法と比べて約 51%に削減された．

参考文献

[1] Android: `http://www.android.com`.
[2] AdMob: `http://www.google.com/ads/admob/`.
[3] 竹森敬祐：スマートフォンからの利用者情報の送信．～情報収集の実態調査～ (2012).
[4] Grace, Michael C. and Zhou, Wu and Jiang, Xuxian and Sadeghi, Ahmad-Reza: Unsafe exposure analysis of mobile in-app advertisements, *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pp. 101–112 (2012).
[5] 川端 秀明, 磯原 隆将, 窪田 歩, 可児 潤也, 上松 晴信, 西垣 正勝：Android における細粒度アクセス制御機構，情報処理学会論文誌， Vol. 54, No. 8, pp. 2090–2102 (2013).
[6] 日置 将太, 齋藤 彰一. 毛利 公一, 松尾啓志：Android における実行コンテキストによるパーミッション切り替え手法の提案，情報処理学会研究報告 2013-CSEC-62， No. 22 (2013).
[7] 日置将太：Android におけるパーミッション適用範囲細分化手法の設計と実装，名古屋工業大学工学部卒業論文 (2012).
[8] PandaBoard ES: `http://pandaboard.org/content/pandaboard-es`.