Computer Security Symposium 2021
26 - 29 October 2021

Androidアプリケーションのテイント解析における例外処理を考慮した制約の精度の向上

大西 導徳[1,a)]　稲吉 弘樹[1]　掛井 将平[1]　齋藤 彰一[1]

概要：マルウェアを用いて機密情報を狙った攻撃が増加している．そのため，アプリケーションからどのような情報がリークするかを検知する手法が必要となる．その一つに静的テイント解析が挙げられる．静的テイント解析は動的テイント解析と比較してコードカバレッジが広い．しかし，検出されたリークが，実際にアプリケーションを動作させても発生しない場合がある．これは，リークがどのような条件が満たされた場合に発生するかという制約を考慮していないことが原因である．ここで，静的テイント解析ツールと組み合わせてリークの制約を求めるツールは存在するが，例外処理のフローについては必ず実行されるものとして扱われており，例外の制約を考慮していない．本稿では例外発生時のフローを考慮した制約条件を求める手法を提案する．既存の制約演算ツールを拡張し，例外が発生する可能性のある命令を検知して，その例外処理のフローに制約を与え，制約条件の精度を向上させる．これにより，例外の制約によってリークが発生しない可能性があることを解析者に示し，リーク対応のコストを下げる．

キーワード：Android，テイント解析，静的解析，制約演算

Improving the Accuracy of Constraints on Exceptions in Static Taint Analysis of Android Applications

Michinori Ohnishi [1,a)]　Hiroki Inayoshi[1]　Shohei Kakei[1]　Shoichi Saito[1]

Abstract: Malware that targets sensitive information is on the rise. Therefore, it is necessary to have a method to detect information leakage from applications. One such method is static taint analysis. It has wider code coverage than dynamic taint analysis. However, in some cases, the leakage does not occur because it does not consider the constraint conditions under which leaks occur. Here, some tools obtain constraint conditions for leaks. Still, they do not consider exception occurrence conditions and treat exception handling flow as always performed flows. This paper proposes a method for obtaining constraint conditions that consider the conditions under which exception handling flows occur. Extending an existing constraint computation tool gets constraint of the exception handling flow and improves accuracy. Our method indicates to the analyst which exceptions may not cause leakage, thereby reducing the cost of leak handling.

Keywords: Android, Taint Analysis, Static Analysis, Constraint Operation

1. はじめに

Android はマルウェアの標的になる可能性が高いといえる [1]．端末内には，端末の識別番号である IMEI や，GPS から取得した位置情報等の機密情報がある．また，利用者自身以外の機密情報に電話帳や SMS の内容等もある．マルウェアの中にはこれらの機密情報を秘密裏に収集しリークするものがある．機密情報がリークされると，個人が特定されたりプライバシーが暴露されたりする上に，新たなサイバー攻撃を受ける危険性がある．そのため，アプリケーションからのリークを検知する方法が必要となる．そ

[1] 名古屋工業大学大学院工学研究科
Graduate School of Engineering, Nagoya Institute of Technology
[a)] clf14155@nitech.jp

の方法の一つにテイント解析がある．

テイント解析とは，制御フローグラフ (Control Flow Graph，CFG) を用いて，プログラム中の変数の値や出力の流れを解析するデータフロー解析技術の一種である．値の流れの解析は，監視対象の値が代入された変数に対してテイントタグと呼ばれる属性情報を付与し，そのテイントタグが付与された変数が読み書きされた場合には，値が伝播したと判断してタグも同時に伝播させることで行う．テイント解析は主に，アプリケーションの解析者がどのような情報がリークしているかを確認し対処するために用いられる．Android アプリケーションを対象とした代表的なテイント解析ツールに TaintDroid[2] と FlowDroid[3], [4] がある．TaintDroid は，Android の仮想化された実行環境を利用して，リアルタイムの解析を行う動的テイント解析ツールである．FlowDroid は，Android ライフサイクルのモデルを用いて解析を行う静的テイント解析ツールである．

FlowDroid は静的に解析を行うため，TaintDroid と比較してコードカバレッジが広いが，誤検知が発生する可能性がある．これは，FlowDroid の検出方法が，テイントタグを最初に付与するメソッドであるテイントソースから，テイントタグが付与された変数が外部にリークするメソッドであるテイントシンクまでの間にフローが存在することを根拠としてリークを検出しているためである．つまり，実際にアプリケーションを実行しても起こりえないフローを検出し，発生しないリークを検出している．

このような誤検知を減らすために，シンボリック実行を用いて制約を求めることができる [5], [6]．制約とは，検出されたリークの発生条件である．制約を求めることで，それぞれのリークがどのような条件が満たされる場合に発生するかを確認できる．TASMAN[5] は FlowDroid の検出結果を用いて，テイントソースとテイントシンクを結ぶフローについてテイントシンクから逆方向にシンボリック実行を行うことで制約を求めて，それらの充足性を SMT ソルバである Microsoft Z3[7] により確認することで，リークが誤検知か否かを確認する．COVA[6] では，FlowDroid の検出結果を用いて，Android 端末の環境設定，ボタン操作等のユーザのインタラクション，入出力操作の 3 種類の操作による制約を Z3 で求め，誤検知か否かの判定を行っている．

しかし，TASMAN と COVA は，例外処理による制約を考慮していない．そのため，例外処理によって発生するリークに対して正しい制約を求めることができない．例えば，あるリークのテイントシンクが例外処理の catch ブロック内に存在する場合，対応する try ブロック内で例外が発生しないと catch ブロックへ実行が遷移しない．このとき，TASMAN と COVA のように例外処理に対する制約を考慮しないと，例外処理は必ず実行されるものと判定される．つまり，実際にアプリケーションを実行した場合に，try ブロック内で例外が発生しなくても，リークは発生すると判定される．より正確な制約を求めるためには，例外処理により発生するリークは，例外が発生した場合にのみ発生するという制約が必要となる．

本稿では，既存のテイント解析ツールを拡張して例外による制約を作成する．これにより，制約の精度を向上させ，発生しないもしくは発生確率の低いリークに対する解析者のリーク発生防止コストを下げる．

本稿の構成は次のとおりである．まず，第 2 章で本研究の関連研究について述べる．第 3 章では提案手法の概要について述べ，第 4 章で既存のテイント解析ツールを拡張する形で行った実装について述べる．第 5 章で提案手法を実装したプログラムでアプリを評価した結果について述べ，第 6 章で現状の提案手法における制限事項について述べ，最後に，第 7 章でまとめる．

2. 関連研究

本章では，テイント解析に関連する研究と，例外処理によるリークが誤検知となる割合を調査した研究について述べる．また，テイント解析で得られた結果から制約を求める研究について述べる．

2.1 テイント解析

テイント解析とは，プログラム中の変数間における値のデータフロー解析技術の一種である．監視対象の値が代入された変数に対してテイントタグと呼ばれる属性情報を付与し，そのテイントタグが付与された変数を監視する．監視対象の値が直接代入された時や監視対象の値がオペランドとして計算された結果が代入された時に，テイントタグを代入先の変数に伝播させることで，監視対象の値の伝播を追跡する．そして，テイントタグが付与された変数が画面に表示されたり外部に出力された場合，監視対象の値がリークされたと判断する．

Android アプリケーションに対してテイント解析を行う研究に TaintDroid がある．TaintDroid は，仮想化された Android の実行環境で実際にアプリケーションを動かす動的テイント解析ツールである．動的に解析を行うため，検出できるリークは実際に実行されたリークに限られ，静的解析に比べてコードカバレッジが狭い．また，マルウェアアプリケーション自身が解析されていることを認識すると，そのアプリがリークを行わないようになり，見逃しの可能性がある [8]．

一方，静的テイント解析は，コードカバレッジが広いという特徴がある．静的テイント解析のシステムに FlowDroid がある．FlowDroid はコンテキスト，フロー，フィールド，オブジェクトを含む Android のライフサイクル全体をモデル化してコールグラフとプロシージャ間制御フローグラフを生成する．事前に定義されたテイントソースとテイント

シンクの間にフローが存在すると，リークとして検出する．

2.2 例外処理によるリーク

King らの研究 [9] では，プログラム中の暗黙的なフローによるリークにおける誤検知について調査している．暗黙的なフローとは，値が直接的に代入されてはいないが，その値が別の手段により移動するフローのことである．6 つのセキュリティ機能に対して調査を行い，検知されたリークの 98%が暗黙的なフローによるものであった．しかし，暗黙的なフローの内 83%は誤検知であった．さらに，97%の暗黙的なフローによるリークは，例外を発生させる可能性があることが判明した．また，通常の条件文によるフローの誤検知率が 30%であるのに対し，例外処理のフローの誤検知率は 85%であると述べている．この研究の結果より，例外処理によって発生するリークはそうでないリークに比べて実際に発生する可能性が低いと言える．

2.3 例外処理と制約

制約とは何らかの事象の発生条件のことであり，条件の成否を真理値によって表す．以下，リークの発生条件をリーク制約，例外処理の発生条件を例外制約，命令文を実行する条件を実行制約と表す．また，複数の制約が関係する複合的な制約を求める処理のことを制約演算といい，制約の真理値による論理演算を行う．本節では，関連研究におけるリーク制約の取り扱い方法について述べる．

FlowDroid に代表される静的テイント解析ツールでは，リーク制約を考慮していないため，実際にアプリケーションを動作させた場合には発生しないリークも検出される．これに対して，FlowDroid で検出されたリークの発生に必要な制約を求める研究に TASMAN がある．TASMAN は，FlowDroid が検出したリークのテイントソースからテイントシンクへのフロー中の実行制約による論理式を生成し，SMT ソルバである Z3 を用いてリーク制約を求める．それが成立しないリーク制約であった場合，そのリークは誤検知であると判定し，FlowDroid が検出したリーク群からそれを取り除くことでリーク検出精度を向上させている．ここで TASMAN の特徴的処理として，フィールドアクセスやメソッド呼び出しの際に，それぞれの基底オブジェクトが NULL でないことを実行制約としている．これは，オブジェクトが NULL となるような実行時エラーは発生しないことを検知の前提としていることを意味する．つまり，TASMAN は例外処理のフローで起こりうるリークについては誤検知判定の対象としていないことが分かる．

FlowDroid が検出したリークに対して制約演算を行う研究に COVA がある．COVA は，バイトコード形式の Android アプリケーションと，事前に定義された制約となりえる API（以下，制約 API）のセットを入力としている．この制約 API には，Android 端末の環境設定，ユーザのインタラクション，入出力操作の 3 種類の要素が含まれており，これらの要素が if 文等の条件で使用された場合に実行制約となる．COVA の処理手順を述べる．最初に，FlowDroid を用いてリークを検出する．次に，CFG を作成し，それに沿って実行制約を求める．ここで，各実行制約をまとめたものを制約マップという．この制約マップは，求めた実行制約から順次追加し，最終的に，プログラム全体の制約マップを作成する．最後に，制約マップからテイントソースとテイントシンクの実行制約を求めて，その論理積をリーク制約としている．また，COVA でも同様に Z3 を用いている．このように，TASMAN では実行されることがないことが明確なフローのみを取り除くのに対し，COVA では制約 API に含まれる 3 種類の要素がどのような条件の下でリークを発生させるかを解析している．しかし，COVA では例外制約を求めていない．つまり，例外処理のフローについては，どのような例外が発生した場合に実行されるのかという例外制約が考慮されていない．

3. 提案手法

本章では，2.2 節で述べた King らの「例外処理によって発生するリークは発生確率が低い」という研究成果を基に，Android アプリケーションの例外制約を求め，リーク制約の精度を向上する手法を提案する．本提案手法により，例外制約を求めることが可能となる．開発者や解析者は，ソースコードがなくてもこれらの例外制約が発生する可能性に基づいて，複数のリークの対応を行う際に例外が出現するリーク制約に対する処置の優先度を下げることができるようになり，対策の効率化を図ることができる．

提案手法の手順は 3 つのステップである．第 1 ステップは，プログラム中の try ブロック内に存在する命令文を抽出する．第 2 ステップは，抽出された命令文により発生する可能性がある例外を特定する．第 3 ステップは，try ブロックに対応する catch ブロックに対して，第 2 ステップで求めた発生する可能性のある例外を例外制約とする．

本章では，提案手法の手順を，図 1 のプログラム例を用いて説明する．このプログラムは，FlowDroid のベンチマークとして作成された DroidBench[10] の Exceptions3.java から一部抜粋したものである．プログラムの動作としては，try ブロック中のテイントソースである getDeviceId メソッドで端末の IMEI を取得して，catch ブロック中のテイントシンクである sendTextMessage メソッドで IMEI を外部に送信するプログラムである．そのため，try ブロック中で例外が発生しないと，sendTextMessage メソッドは実行されない．

3.1 try ブロック中の命令文の抽出

第 1 ステップとして，try ブロック中 (図 1 の 3〜8 行目) で例外が発生した場合に catch ブロックへ実行が遷移する

```
1  String imei = "";
2  try {
3    TelephonyManager telephonyManager =
4    (TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE);
5    imei = telephonyManager.getDeviceId(); //source
6    int[] arr = new int[42];
7    if (arr[32] > 0)
8      imei = "";
9  }
10 catch (RuntimeException ex) {
11   SmsManager sm = SmsManager.getDefault();
12   sm.sendTextMessage("+49 1234", null, imei, null, null); //sink, leak
13 }
```

図 1 Exceptions3.java(一部抜粋)
Fig. 1 Exceptions3.java (partial excerpt)

可能性のある命令文を抽出する．例外処理は，try ブロック中で例外が発生した後，対応する catch ブロックに実行が遷移することで発生するためである．抽出には，COVA で使われている CFG を用いる．この CFG には，命令文毎に発生する可能性がある例外と，その例外が発生した場合にどの命令文に遷移するかという情報がある．それらの情報を使用して，try ブロック中の命令文を抽出する．

3.2 発生する可能性のある例外の調査

第 2 ステップとして，try ブロック中の命令文において発生する可能性がある例外を調べる．これは，catch ブロックに遷移する例外の種類と，例外を発生させる命令文の実行条件から，例外処理が発生する条件，つまり例外制約を求めるために行う．

図 1 のプログラム例を用いて，調査方法を述べる．4 行目では，getSystemService メソッドの返り値を TelephonyManager クラスにダウンキャストしている．そのため，ClassCastException が発生する可能性がある．5 行目の telephonyManager.getdeviceId() では，仮想メソッド呼び出しが行われており，telephonyManager が null である可能性がある．そのため，NullPointerException が発生する可能性がある．6 行目では，配列の宣言が行われているため NegativeArraySizeException が発生する可能性がある．しかし，図 1 では配列のサイズは 42 となっており，負の値ではないため NegativeArraySizeException が発生する可能性はない．7 行目では，配列にアクセスされているため，ArrayIndexOutofBounsException が発生する可能性がある．しかし，図 1 では，配列 arr の宣言時のサイズは 42 であり，アクセス位置は固定で 32 であるため ArrayIndexOutofBoundsException は発生しない．8 行目では String 型の変数 imei に空白が代入されているだけなので，例外は発生しない．よって，try ブロックで発生する可能性のある例外は ClassCastException と NullPointerException となる．例外が発生した場合の遷移先は，いずれも cactch ブロックとなる．

3.3 例外制約の作成

第 3 ステップとして，発生する可能性のある例外を catch ブロック内の命令文に対する例外制約とする．3.2 節で求めた発生する可能性のある例外が発生すると，catch ブロックに実行が遷移する．そのため，“ClassCast or NullPointer” という例外の論理式で例外制約を作成し，catch ブロックに対してこの例外制約を付与する．

4. 実装

本章では，提案手法の実現のため，既存の制約演算ツールを拡張し，catch ブロックに例外制約を付与する方法について述べる．

4.1 既存の制約演算ツールの拡張

FlowDroid と TASMAN と COVA は，Java と Android アプリケーションの分析を行うフレームワークである Soot[12] を拡張する形で実装されている．Soot には，プログラムを Jimple[12] と呼ばれる中間表現で表した CFG を作成する機能があり，FlowDroid と TASMAN，COVA で使用されている CFG も Soot で作成されたものである．

FlowDroid の検知結果からリーク制約を演算する既存のツールは，2 章で述べた TASMAN と COVA があるが，本研究では COVA を拡張して catch ブロックに対する例外制約を求める．COVA を拡張する理由は，プログラム全体の実行制約を既に求めているためである．これにより，テイントソースからテイントシンクの間の実行制約だけでなく，テイントソースの実行に必要な実行制約を求めることができる．また，COVA のソースコードは github 上で公開されており，拡張が容易な点も理由である [11].

4.2 実装システムの概要

図 2 に実装システムの全体像を示す．システムは，FlowDroid と拡張した COVA を併用して実装している．まず，1) バイトコード形式に変換した Android アプリケーションを元に，FlowDroid でリークを検出する．次に，2) 拡張し

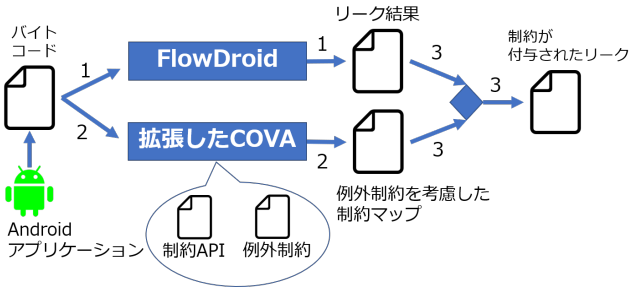

図 2　実装システムの全体像

Fig. 2　Overall view of the implementation system.

た COVA は，作成した CFG に沿って，プログラム全体に対してそれぞれの命令文を実行するために必要な制約マップを求める．拡張した COVA が対象とする制約は，事前に定義された 3 種類の要素の制約 API と，例外制約である．本提案手法によって拡張した COVA は，制約マップを作成する処理において，例外制約を加える．以上の手順で，例外制約を考慮した制約マップを作成している．最後に，3) リーク結果と制約マップから，各リークのテイントソースとテイントシンクの実行制約を取得して出力する．

4.3　try ブロック中の命令文の取得

提案手法の第 1 ステップの実装では，COVA の解析が新しいコンテキストに到達した場合に，CFG を取得する．CFG には，例外が発生する可能性のある命令文，すなわち，try ブロック中の命令文と，その例外が発生した場合の遷移先の命令文の情報がある．それらの情報を取得する．

4.4　発生する可能性のある例外の種類の特定

提案手法の第 2 ステップの実装として Soot を用いる．Soot には，ある命令によってスローされる可能性のある Runtime Exception とエラーのセットを返すクラスが実装されている．そのクラスを用いて，どのような Runtime Exception が発生する可能性があるかを調べる．このとき，コンテキスト中で同じ Runtime Exception が発生する可能性がある．それらの例外を区別するために番号付けを行う．COVA の解析中のコンテキストの番号と，その例外がコンテキスト中の出現回数で区別する．なお，対応する例外は次節で述べる．

さらに，try ブロック中に存在する throw 文によって，実行が catch ブロックに遷移する場合がある．この場合，throw 文の実行制約を制約マップから取得し，その実行制約を catch ブロックへ遷移するための例外制約とする．

4.5　対応する Runtime Exception

第 2 ステップで対応している例外は，Soot で対応済みの例外である．Soot が Android アプリケーションを解析する場合に対応している Runtime Exception は，ArithmeticException，ClassCastException，IllegalMonitorStateException，NegativeArraySizeException，ArrayIndexOutOfBoundsException，NullPointerException の 6 種類である．これらの内，Soot での実装に加えて，独自に拡張した機能について述べる．

4.5.1　NegativeArraySizeException

この例外が発生する可能性について，Soot では，変数または 0 より大きい整数で配列が作成されている場合にこの例外が発生する可能性があるとしている．つまり，Soot ではこの例外が発生する可能性のある条件に誤りがある．そこで，変数または 0 より小さい整数で作成されている場合にこの例外が発生する可能性があるように修正を行った．なお，この誤りについて連絡し，現在は修正が完了している [13]．

4.5.2　ArrayIndexOutOfBoundsException

この例外が発生する可能性について，Soot では，配列にアクセスした場合にこの例外が発生する可能性があるとしている．これは，配列アクセスを行う命令文のみではその配列の最大サイズを取得できないためである．そこで，COVA の解析中に配列を作成する命令を検出した場合，配列のサイズを記録する拡張を行った (配列のサイズが変数となっている場合，サイズを 0 として扱っている)．配列アクセスを行う命令を検出した場合，その配列のサイズを取得し，そのインデックスが配列の範囲外または変数となっている場合にのみこの例外が発生する可能性があるように拡張を行った．

4.5.3　NullPointerException

この例外が発生する可能性について，Soot では，インスタンスメソッドの呼出し，フィールドアクセス，配列アクセス，配列の長さの取得，Throw 文の実行，モニタの使用を行う場合にこの例外が発生する可能性があるとしている．また，Soot には，ある地点においてある変数の値が，「確実に null である」，「確実に null でない」，「どちらでもない」の 3 種類のどの状態かを調べるクラスがある．そこで，このクラスを用いて，ベースオブジェクトや配列が確実に null である，またはどちらでもない場合にのみこの例外が発生する可能性があるように拡張を行った．

4.5.4　その他の例外

Android の Runtime Exception には多くの Exception があるが，現状で Soot がサポートしている例外は本節で述べた 6 種類の Runtime Exception のみであり，これら以外の例外を検知することはできない．そこで，catch ブロックに付与する例外制約に OtherExceptions という例外制約を付与する．これは，try ブロック中に存在する可能性がある，Soot では検知できない例外を表している．そのため，この例外制約は catch ブロックに対して必ず付与される．この例外制約を付与することで，6 種類の Runtime Exception 以外の例外によって例外処理に遷移する可能性があること

を解析者に伝えている．なお，try ブロック中で Soot では検知できない例外が存在しない場合もあるため，6 種類の Runtime Exception よりも，リーク対応の優先度は低いと考える．

4.6 例外制約の組み込み

提案手法の第 3 ステップの実装として，例外制約を制約マップに組み込み，catch ブロックに対する例外制約として付与する．COVA の処理中に，catch ブロックの始まりを意味する命令文に到達すると，第 2 ステップで求めた，対応する try ブロックで発生する可能性のある例外と，throw 文の実行に必要な実行制約を，Z3 で扱える論理式として作成し，それらの論理和を求める．これは，try ブロック中で発生する可能性のある例外か発生するか，throw 文が実行された場合に，catch ブロックへ実行が遷移するためである．作成した論理式と，catch ブロックの始まりを意味する命令文の実行制約を制約マップから取得し，それらの論理積を求める．最終的に求めた論理式を COVA の制約マップに組み込む．これにより，catch ブロックに例外制約を付与する．

4.7 提案手法の出力結果

COVA は制約マップの作成が終了すると，FlowDroid で検出されたリークのテイントソースとテイントシンクがどのクラスのどのメソッドに存在するかという情報と，制約マップからそれぞれの実行制約と，それらの実行制約の論理積，すなわち，リーク制約を出力する．リーク制約に例外が含まれる場合，そのリークは本提案手法で追加された制約であり，例外処理のフローによって発生することを意味する．このようなリークは，例外制約が真，すなわち，必ず発生するリークよりも実際に発生する可能性が低いと考えられるため，解析者はこのようなリークに対応する優先度を下げることで，どのリークから対応を行うかを決定する時間的コストを削減できる．

5. 評価

本章では提案手法の評価方法とその結果について述べる．今回の評価で用いた評価環境を表 1 に示す．

5.1 評価方法

提案手法を実装した COVA を用いて，検出されたリーク制約に含まれる例外の数を確認した．また，種類別の例外数を，4.5 節で述べた Runtime Exception を対象として確認した．

5.1.1 解析対象のデータセット

評価には 2 種類のデータセットを解析対象とした．1 つ目のデータセットは COVA Dataset[14] の analyzed フォルダにある 315 個の Android アプリケーションである．これらは，COVA の製作者の解析環境において，30 分以内に解析が完了したものである [6]．2 つ目のデータセットは Baidu(百度手机助手 [15]) から 2021 年 7 月 15 日に収集したものである．まず，5 つのランキング [15] からアプリ個別ページの URL を集めて，重複を除いた 248URL からアプリをダウンロードした．それらのファイルを確認して，zip ファイルと，重複したアプリを削除した．その結果，222 個のアプリケーションを対象とした．

表 1 評価環境
Table 1 Environment.

CPU	Intel Core i7-10700K 3.80GHz
OS	Ubuntu20.04LTS
FlowDroid	2.5.1
Microsoft Z3	4.5.0
JVM の最大ヒープサイズ	64GB

図 3 COVA のタイムアウトの処理
Fig. 3 Handling COVA timeouts.

5.1.2 解析条件

FlowDroid は事前にテイントソースとテイントシンクを指定する必要がある．本研究の評価で使用したテイントソースは 35 個で，テイントシンクは 121 個である．これは，FlowDroid2.5.1 のデフォルトの設定から，COVA の論文 [6] で示されている不適切な 11 個のテイントソースと 1 個のテイントシンクを除外したものである．COVA が対象とする制約 API は COVA の論文と同じものを用いた．

COVA には 2 種類のタイムアウトがある．それぞれを図 3 に示す．1 つ目のタイムアウトは，COVA のみの処理に対するタイムアウトであり，30 分である．2 つ目のタイムアウトは，前処理の FlowDroid と後処理の結果出力を含めたタイムアウトであり，合計で 60 分である．COVA の実行時間が 30 分を経過した場合，その時点での制約マップを用いてリーク制約の取得を行い，解析結果を出力する．一方，解析プロセス全体の実行時間が 60 分を経過した場合，プロセスを強制的に終了する．この場合，提案手法での解析結果を取得することはできない．また，解析途中でエラーが発生した場合も同様に解析結果を取得することはできない．

5.2 評価結果

評価結果について述べる．

5.2.1 各データセットの解析結果

表 2 にそれぞれのデータセットにおける解析完了アプリ数を示す．「リークなし」は，FlowDroid でリークが検出

表 2　各データセットの解析結果 (アプリ数)
Table 2　Analysis results for each data set.

データセット	アプリ総数	リーク		リークありアプリの解析結果			
		なし	あり	完了	COVA タイムアウト	全処理タイムアウト	失敗
COVA	315	111	204	87	116	0	1
Baidu	222	81	141	1	103	7	30

表 3　リーク数と例外制約を含むリーク数
Table 3　Number of leaks with exception constraints.

データセット		アプリ		リーク		
		(アプリ数)	例外を含む	(リーク数)	例外を含む	制約 API を含む
COVA	解析完了	87	9	800	32	25
	COVA タイムアウト	116	21	1779	139	78
Baidu	解析完了	1	0	2	0	0
	COVA タイムアウト	103	15	5384	36	5

されなかったアプリ数である．これらは COVA による解析の対象外とする．よって，COVA データセットでは 204 個，Baidu データセットでは 141 個のアプリケーションを解析対象とする．「解析完了」と「COVA タイムアウト」が示すアプリ数が，最終的に結果を取得したアプリ数である．「解析完了」が示すアプリは，プログラム全体の制約マップを取得できたのに対し，「COVA タイムアウト」が示すアプリは，一部のプログラムの制約マップのみを取得できた．結果を取得できたアプリにおける COVA タイムアウトの割合は，COVA データセットでは約 57.1%が，Baidu データセットでは約 99.0%である．Baidu データセットの方がタイムアウトになる割合が多いのは，Baidu データセットのアプリのサイズが COVA データセットにあるアプリのサイズよりも大きい傾向にあることが原因であると考える．

5.2.2　例外制約を含むリーク数

表 3 にそれぞれのデータセットで検出されたリークと，例外制約を含むリーク数を示す．COVA データセットでは，合計 30 個のアプリで例外制約を含むリークがあり，171 リークに例外制約が出現した．一方，Baidu データセットでは，合計で 15 個のアプリで例外制約を含んだリークがあり，36 リークに例外制約が含まれた．これらのリークは，例外処理のフローによって発生するため，リーク対応の優先度を下げることができる．

また，「制約 API を含む」が示すアプリは，制約 API と例外制約の両方を含むリークのことで，COVA データセットでは合計で 103 リーク，Baidu データセットでは合計で 5 リークのリーク制約に制約 API も含まれていた．これらのリークは，両方のリーク制約を満たさないと発生しないので，どちらか片方の制約しか出現しないリークよりも発生する可能性が低いと考えられるため，リーク対応の優先度をより下げることができる．

5.2.3　例外別のリーク数

表 4 にそれぞれのデータセットのアプリにおける，例外別のリーク数を示す．OtherExceptions は，どの catch ブロックに対しても付与するために，例外制約があれば必ず出現する．どちらのデータセットにおいても，NullPointerException が OtherExceptions を除く 6 種類の Exception の中で最も多く出現した．これは，メソッド呼び出しやフィールドアクセス等が頻繁に行われるためと考える．

5.3　評価結果のまとめ

評価の結果，COVA データセットではリークの約 6.6%，Baidu データセットではリークの約 0.7%に例外制約が出現し，NullPointerException が最も多く出現した．これらの例外処理のフローによって発生するリークは，対応の優先度を下げ，他のリークを優先的に対応することができる．

6.　制限事項

COVA で使用されている FlowDroid2.5.1 は 2018 年 1 月 17 日にリリースされており [16]，最新のアプリケーションに対応していない可能性があり，一部のアプリケーションで解析に失敗した原因であると考えられる．また，FlowDroid は非決定的であることが示されている [17]．そのため，FlowDroid が複数回同じ構成で同じアプリケーションを解析しても，毎回同じ結果が出力されるとは限らない．

「COVA タイムアウト」となる主な要因は，COVA の制約演算に使用されている Z3 である．結果を出力できたアプリの解析時間の内，COVA データセットでは平均 63.7%が，Baidu データセットでは平均 63.5%が Z3 によって占められていた．処理の高速化は今後の課題である．

図 1 のプログラムでは，4 行目で ClassCastException または 5 行目で NullPointerException が発生した場合に，catch ブロックに実行が遷移する．しかし，getDeviceId メ

表 4　対応する例外を含むリーク数
Table 4　Number of leaks by exception.

	COVA データセット		Baidu データセット	
例外の種類	解析完了	COVA タイムアウト	解析完了	COVA タイムアウト
例外制約を含むリーク総数	32	139	0	36
NullPointer	10	123	0	29
ClassCast	5	17	0	1
ArrayIndexOutOfBounds	0	0	0	7
NegativeArraySize	0	0	0	0
IllegalMonitorState	0	0	0	0
ArithmeticException	0	0	0	0
OtherExceptions	32	139	0	36

ソッドが実行されていないため変数 imei には IMEI の値が格納されていない．よって，sendTextMessage が実行されても IMEI の値はリークしない．そのため，テイントソースである getDeviceId メソッドの実行制約に，これらの例外が発生しないことを実行制約として追加する必要があり今後の課題である．

Soot が Android アプリケーションを解析する場合に対応している 6 種類の Runtime Exception は，命令のみでその例外が発生する可能性があるかを判断できるために対応していると考えられる．本来は入出力に関するような Exception 等も扱うべきであるが，API そのものの解析が必要になるため，今後の課題とする．

7. まとめ

本研究では，例外処理のフローに対する制約を考慮した制約を求め，リーク制約の精度を向上する手法を提案した．既存の制約演算ツールである COVA を拡張して，例外制約を求める実装を行った．

提案手法を実装したツールを用いて実アプリの解析を行い，リーク制約に例外が出現するリーク数の評価を行った．その結果，データセットで検出できた全リークの約 2.6%で例外制約が出現した．これらのリークは，例外制約が出現しないリークよりも発生する可能性が低いと考えられるため，リーク対応の優先度を下げることが可能となり，解析者が他のリークを優先して対応できるようになる．

今後の方針として，アプリケーションの解析時間の短縮を試みる．また，扱う例外の種類を増やす．

謝辞　本研究を遂行するにあたり，COVA の開発者である Linghui Luo 氏と FlowDroid の開発者である Steven Arzt 氏に厚く御礼を申し上げる．

参考文献

[1] 乙部幸一朗：なぜモバイル向けマルウェアの98%はAndroidをターゲットとしているのか，入手先 ⟨https://japan.zdnet.com/article/35063552/⟩ (2021.07.21).
[2] Enck, W., Gilbert, P., Han, S., Tendulkar, V., and Chun, B., Cox, L. P., Jung, J., McDaniel, P. and Sheth, A. N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones, ACM Transactions on Computer Systems (TOCS) (2014).
[3] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D. and McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, Acm Sigplan Notices (2014).
[4] Arzt, S.: Static Data Flow Analysis for Android Applications, (2017).
[5] Arzt, S., Rasthofer, S., Hahn, R. and Bodden, Eric.: Using targeted symbolic execution for reducing false-positives in dataflow analysis, Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis (2019).
[6] Luo, L., Bodden, E. and Späth, J.: A qualitative analysis of Android taint-analysis results, 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (2019).
[7] Microsoft Z3, 入手先 ⟨https://github.com/Z3Prover/z3⟩ (2021.07.21).
[8] Percoco, N. J. and Schulte, S.: Adventures in bouncerland, Blackhat USA (2012).
[9] King, D., Hicks, B., Hicks, M. and Jaeger, Trent.: Implicit flows: Can't live with 'em, can't live without 'em, International Conference on Information Systems Security (2008).
[10] DroidBench, 入手先 ⟨https://github.com/secure-software-engineering/DroidBench⟩ (2021.07.06).
[11] COVA, 入手先 ⟨https://github.com/secure-software-engineering/COVA⟩(2021.07.07).
[12] Lam, P., Bodden, E., Lhoták, O. and Hendren, L.: The Soot framework for Java program analysis: a retrospective, Cetus Users and Compiler Infastructure Workshop (CETUS 2011).
[13] Fixed wrong, 入手先 ⟨https://github.com/soot-oss/soot/commit/fa3e1737f42619aa6dfa18a7080a4485e38e9e18⟩ (2021.07.13).
[14] COVA Dataset, 入手先 ⟨http://www.ipsj.or.jp/journal/submit/manual/j_manual.html⟩ (2021.07.06).
[15] 百度手机助手，
入手先 ⟨https://shouji.baidu.com/rank/top⟩
入手先 ⟨http://as.baidu.com/rank/top/software⟩
入手先 ⟨http://as.baidu.com/rank/top/game⟩
入手先 ⟨http://as.baidu.com/rank/up⟩
入手先 ⟨http://as.baidu.com/rank/features/classic⟩
(2021.08.11).
[16] FlowDroid-releases, 入手先 ⟨https://github.com/secure-software-engineering/FlowDroid/releases⟩(2021.07.14).
[17] Benz, M., Kristensen, E. K., Luo, L., Borges, N. P., Bodden, E. and Zeller, A.: Heaps'n Leaks: How heap snapshots improve android taint analysis, IEEE/ACM 42nd International Conference on Software Engineering (ICSE) (2020).