特別研究報告

題目

ポインタ解析におけるライブラリのスタブコードへの置換の効果

指導教員
井上 克郎 教授

報告者
山本 佑也

平成 29 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 28 年度 特別研究報告

ポインタ解析におけるライブラリのスタブコードへの置換の効果

山本 佑也

内容梗概

プログラム中で利用されるデータの流れを追跡するデータフロー解析の 1 つとしてポインタ解析がある．ポインタ解析とはプログラム中の各々の変数について，その変数にどのようなオブジェクトが代入される可能性があるのかを解析する手法である．ポインタ解析において，あるプロジェクトが利用している外部ライブラリをどう扱うかというのは大きな問題である．ライブラリはしばしば多大な解析時間やメモリ使用量を要するほどの容量であったり，解析が困難なネイティブコードを含んでいるなど素直に解析をするには障害となる要素が存在する．その問題を回避する方法としては，ライブラリの一部のみを解析するという方法も存在する．しかし，ライブラリで直接呼び出した機能の内部で，さらに別の機能を利用している場合などもあり，解析すべきライブラリの一部分の特定が困難である．現時点で提案されている手法としては，個々の解析ごとにその解析内での外部ライブラリの扱いについて記述したサマリーを用意し，解析時に外部ライブラリそのものを解析する代わりにサマリーを適用するというものがある．

本研究では解析ごとのサマリーから一歩進んで，解析手法に依存しないサマリーの表現としてのスタブコードを提案し，Java の複数のプロジェクトに対するポインタ解析における手法の有効性を評価した．その結果，外部ライブラリ中の特定の一部についてのスタブコードを作成するだけで，外部ライブラリを完全に無視した場合と比べて精度を大きく向上させられることを確認した．

主な用語

ポインタ解析
スタブコード
外部ライブラリ

目 次

1　前書き **3**

2　背景 **4**

2.1　ポインタ解析 4

2.2　ポインタ解析における課題 5

2.3　先行研究における外部ライブラリの扱い 7

3　提案手法 **9**

3.1　外部ライブラリのスタブコード表現 9

3.1.1　Java におけるスタブコード例 9

3.1.2　クラスの表現 9

3.1.3　変数・フィールドの表現 11

3.1.4　メソッドの表現 11

3.2　ポインタ解析への適用 11

3.3　スタブコードの利点 12

4　評価実験 **14**

4.1　実験方法 14

4.2　評価方法 14

4.3　実験結果 16

4.4　考察 17

5　手法の拡張に向けた議論 **21**

5.1　スタブコードの追加 21

5.2　他のデータフロー解析への適用 21

6　まとめ **23**

謝辞 **24**

参考文献 **25**

1　前書き

プログラム中で利用されるデータの流れを追跡するデータフロー解析 [11] の 1 つとしてポインタ解析がある．ポインタ解析とはプログラム中の各々の変数について，その変数にどのようなオブジェクトが代入される可能性があるのかを解析する手法である．ポインタ解析において，あるプロジェクトが利用している外部ライブラリをどう扱うかというのは大きな問題である．ライブラリはしばしば多大な解析時間やメモリ使用量を要するほどの容量であったり，解析が困難なネイティブコードを含んでいるなど素直に解析をするには障害となる要素が存在する．その問題を回避するためにライブラリの一部のみを解析するという方法も存在する．しかし，ライブラリで直接利用した機能の内部でさらに別の機能を利用している場合などもあり，解析すべきライブラリの一部分の特定が困難である．

Arts ら [2] によれば，現在の Android アプリケーションに対するデータフロー解析において，外部ライブラリの扱いには主に下記の 3 つのパターンが存在している．

1. 外部ライブラリを全て含めてそのまま解析をする [9]
2. 「あるメソッドの戻り値はその引数の全てと関係がある」等の規則を定義し，その規則に基いて外部ライブラリを扱う（外部ライブラリ自体は解析しない）[7]
3. 各解析において必要な情報を外部ライブラリの中から人力でサマリー（要約）を作り，それを解析に利用する [3][4][5][6][8][10][13]

これらの手法を一般的な外部ライブラリに適用する場合，1 つ目の手法ではライブラリの規模によっては多大な時間が必要となり，2 つ目の手法では解析の精度に問題が出る可能性がある．一方で，精度や所要時間の面で両者の中間的な存在である 3 つ目の手法については，それぞれの実験で作成されるサマリーの表現方法は独自性の高いものであり，解析ごとに新たなサマリーの作成と，その利用アルゴリズムの実装を行う必要がある．

本研究ではその 3 つの手法のうちの 3 つ目に焦点を当て，解析手法に依存しないサマリーの表現方法としてのスタブコードと，解析対象のソフトウェアとスタブコードに同時に解析を適用する手法を提案する．以降，2 章では本研究の背景について詳しく述べる．3 章では本研究で提案するスタブコードを用いた手法について紹介し，4 章ではその評価実験について述べる．5 章では，手法の拡張に向けた議論を行い，6 章で本研究のまとめと今後の課題を述べる．

2　背景

2.1　ポインタ解析

ポインタ解析とは，ソースコード内の変数においてその変数がコード上のどこで生成されたオブジェクトが代入されうるかを解析するデータフロー解析の一種である．

ポインタ解析でも基本的なアルゴリズムの 1 つに Andersen の提案したポインタ解析アルゴリズム [1] がある．このアルゴリズムはどの順番で代入文が実行されるかという制御フローを考慮していない (flow-insensitive) アルゴリズムである．

Andersen のポインタ解析アルゴリズムの流れは以下の通りである．

1. オブジェクトの生成箇所を探す
2. そのオブジェクトが変数に代入されている箇所を探し，代入されている変数からオブジェクトを指す有向辺を生成する
3. 別の変数からある変数に代入されている箇所を探し，代入されている変数から代入した変数が指している全てのオブジェクトを指す有向辺を生成する
4. 全ての代入箇所に対して 2,3 を繰り返す
5. 2-4 の手順を新たに有向辺が生成されなくなるまで繰り返す

以上の手順を行うと変数とオブジェクトを頂点とし，変数をその変数に代入されうるオブジェクトへの有向辺で繋いだ有向グラフができる．このグラフを本研究ではポインタグラフと呼ぶ．ポインタグラフの構築により，それぞれの変数に対して，その変数に代入されうる全てのオブジェクトを網羅することがポインタ解析の目的である．

図 1 は擬似コードに対してポインタ解析を実行した流れを示している．1 段目は手順 1,2 を表している，a と b に対してそれぞれ Hoge と Fuga のオブジェクトを代入しているので，a から Hoge のオブジェクトへ，b から Fuga のオブジェクトへと有向辺を引いている．2 段目は手順 3 を表している．a から c への代入式，b から d への代入式を処理している．それぞれ a と b から有向辺が引かれている Hoge,Fuga のオブジェクトに対して有向辺を引いている．b から d への代入は，if 文の条件式の評価によっては実行されない可能性があるが，ポインタグラフは代入されうる可能性のある全てのオブジェクトに対して有向辺を引くため，制御文の有無に関わらず有向辺を引く．3 段目は手順 4 を表している．c から d への代入式があるため，d から c が指している Hoge のオブジェクトに有向辺を引く．d は Hoge と Fuga の両方を指しているが，代入されうる全てのオブジェクトに有向辺を引くため，このように 2 つ以上の有向辺が引かれる場合もある．そして d から e への代入式があるため，e

は d と同様に Hoge と Fuga のオブジェクトに対して有向辺を引いている．手順 5 では新しい辺が引かれなくなるまで全ての代入文を繰り返しチェックする．今回のケースでは一周するだけで全ての有向辺を引くことが出来た．もし代入文に変数が登場する順序が複雑な場合は複数回チェックを行うことになる．

また，実際のポインタ解析ではメソッド呼び出しやオブジェクトのフィールドについても考慮される．フィールドについては，まずどのオブジェクトのフィールドかを特定した上で，そのフィールドからオブジェクトに対して有向辺を引く．メソッドについては，メソッドの戻り値が，メソッドの引数や変数とどういう関係にあるのかを解析し，それをもとに戻り値になりうるオブジェクトを特定し，それらへの有向辺が生成される．図 2 はオブジェクトのフィールドへの代入の例である．e のフィールド num に対して Obj のオブジェクトが代入されている．この場合，まず e が何を指すのかを現在構築されているポインタグラフから特定する．e は Hoge と Fuga の両方を指しているので，e.num とは Hoge のオブジェクトのフィールド num，Fuga のオブジェクトのフィールド num のどちらを指している可能性もある．両方のオブジェクトのフィールド num に Obj のオブジェクトが代入される可能性があることを表現するため，それぞれの num から Obj のオブジェクトに対して有向辺を引く．

2.2 ポインタ解析における課題

ポインタ解析において，解析対象に外部ライブラリが含まれているとその扱いが難しくなる．外部ライブラリは複数の機能がまとめて実装されていることが一般的であり，クライアントサイドでは利用されていない機能を多く含んでいる可能性が高い．クライアントサイドから利用されている部分のみを解析対象に含めようとしても，内部的に複雑な呼び出し階層になっているなど対象部分の切り出しが困難な場合もある．それを回避するために外部ライブラリを丸ごと解析対象に含めてしまうと解析時間やリソースの使用量と言った解析コストが高くなりすぎる恐れがある．また，Java のネイティブメソッドなどそもそも対象言語のプログラムとして実装されておらず，解析が困難な場合も存在する．一方で外部ライブラリを無視した場合，内部のポインタグラフが構築できないためにメソッドの引数や戻り値，内部のフィールドの関係が掴めない．そのため外部ライブラリのデータフローに関係する有向辺の生成が行えず，解析の精度に問題が出てしまう．図 3 では，Hoge が外部ライブラリのオブジェクトである場合に起こる事態を表している．Hoge の含まれるライブラリについて解析出来ていれば，ポインタグラフ中に破線で書かれている有向辺と薄い青色で示されている val, arg についての情報が得られる．しかし，そのライブラリを解析しない場合は，f に代入されている Hoge#get の戻り値が Hoge オブジェクトのフィールド val であることや，val がどのオブジェクトを指しているのかなどの f からの有向辺を生成するのに必要な情報が得られない．そのため，実際には f は Fuga のオブジェクトに対して有向辺を引く必要がある

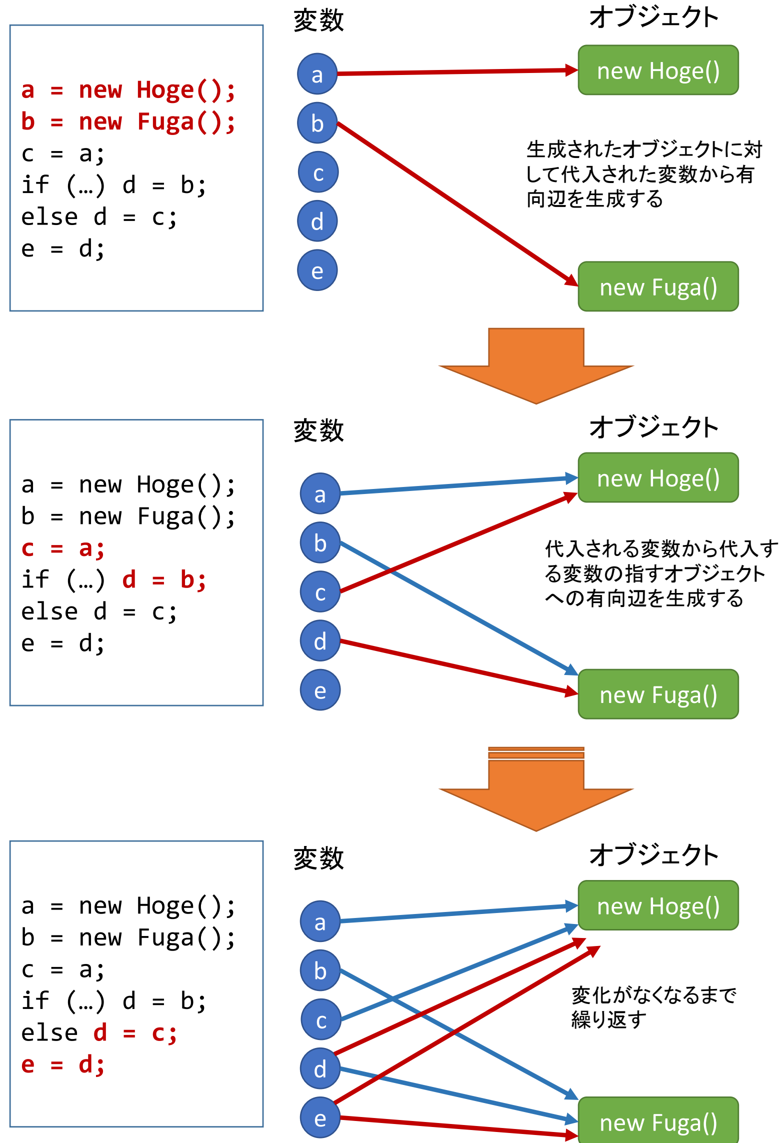

図 1: ポインタ解析の流れ

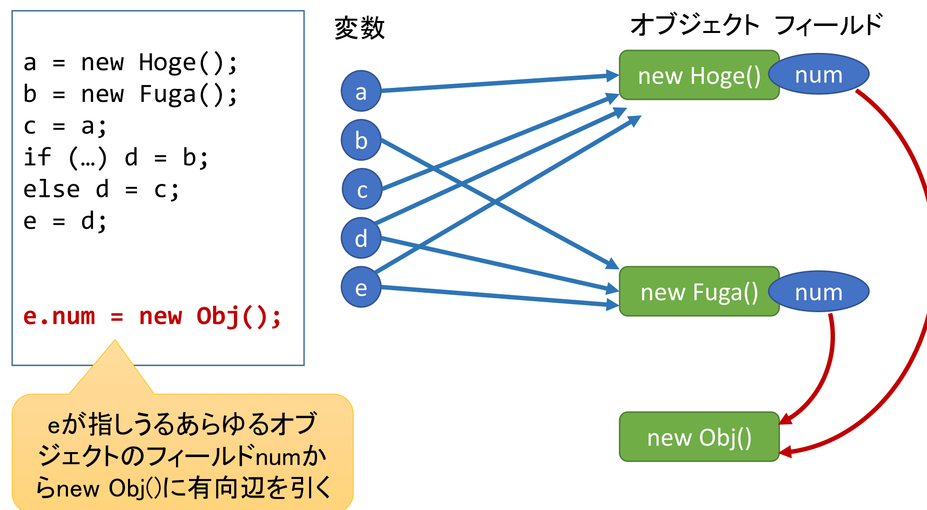

図 2: ポインタ解析でのフィールドの扱い

にも関わらず，その有向辺を引くことが出来ない．

2.3 先行研究における外部ライブラリの扱い

Arzt ら [2] は Android のソフトウェアに対する Taint Analysis の実装をする際，解析に関わる外部ライブラリの解析の代わりとなる情報を予め解析することで準備し，XML 形式で表現する手法を提案している．Taint Analysis とはプログラム中の特定のデータ（パスワード等の機密情報など）に対して taint(汚染) という情報を付加し，それが伝播する法則を設定してそのデータがどこまで伝播するかを調べるデータフロー解析の一種である．ソースコードを解析することで，オブジェクトのフィールドやメソッドの引数，戻り値などにどのように taint が伝播するかという情報を構築する．それに基づいてある特定のデータについた taint がどのようなフィールドやメソッドを通してどこまで伝播するか特定する．しかし，外部ライブラリ内のオブジェクトに対しては解析が行われないため，伝播の情報が構築されない．そこで Arzt らはあらかじめ外部ライブラリに関係する taint の伝播の規則をまとめたサマリーを用意しておき，解析の際にはそのサマリーを元に伝播の有無を判定するという手法を用いている．ただし実際に外部ライブラリを解析した際に構築される伝播情報を，伝播の規則という形で一度 XML に表現するため，直接解析に含めた場合と解析の挙動に差異が出ないように工夫を要していた．

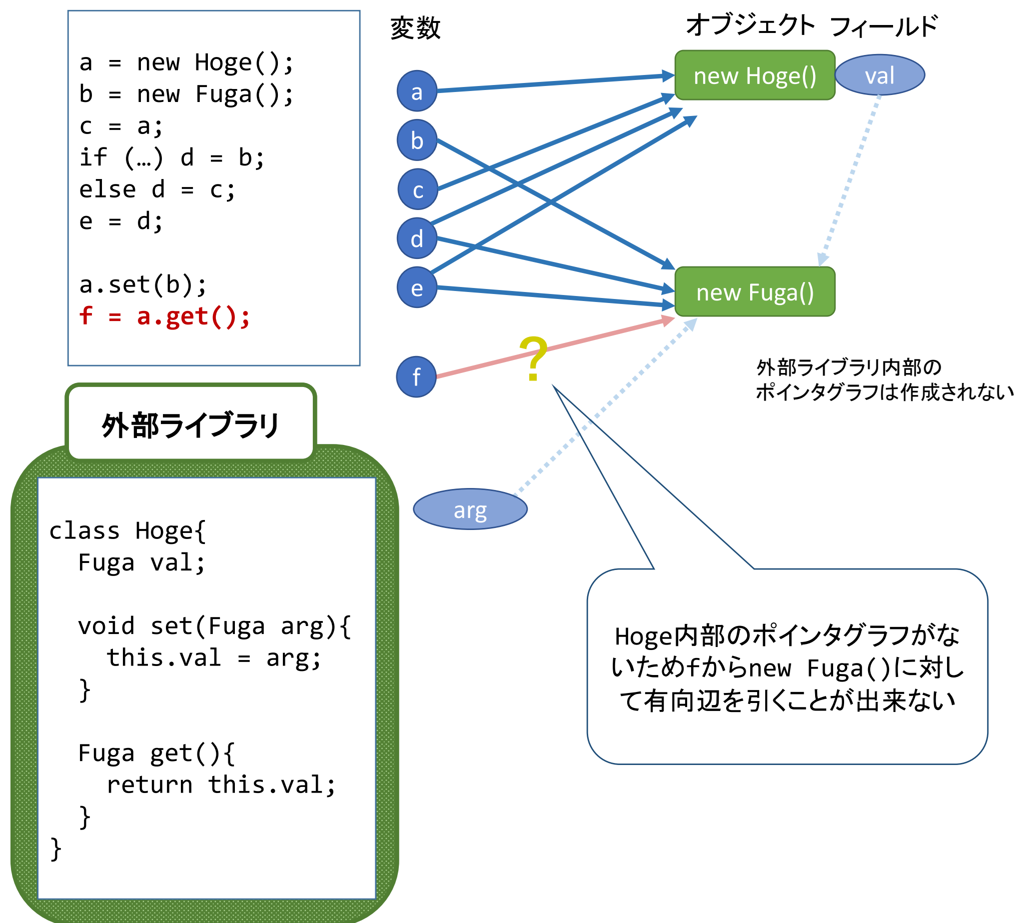

図 3: 外部ライブラリを解析しない場合の問題点

3 提案手法

本章では提案手法を説明する．本手法では，外部ライブラリを直接解析対象とする代わりに，外部ライブラリ内のメソッドや変数に関するポインタグラフを解析に必要な範囲で再現できるようにスタブコードと呼ばれる簡略化したソースコードを作成し，解析に適用する．

次節からはスタブコードの作成と，その解析への適用方法についてそれぞれ詳しく述べる．

なお，本章以降では，特定のクラスの特定のメソッドを指す際，クラス名#メソッド名 という表記を用いる

3.1 外部ライブラリのスタブコード表現

ライブラリ内のオブジェクトの流れを表すスタブコードを作成する．スタブコードとは本来はソフトウェアのテストなどの際に未完成部分の代替として想定される振る舞いと似た挙動を示すコードのことである．ここでは外部ライブラリのオブジェクトへのデータの流入とそのメソッドからの戻り値やフィールドからのデータの流出のみに焦点を絞ったコードを指す．以降，まずは例となる Java のスタブコードを示し，その後ライブラリ内の機能ごとのスタブコードの書き方を説明する．

3.1.1 Java におけるスタブコード例

図 4 は評価実験に際して作成した Java クラスライブラリの java.util.Collection インターフェースのスタブコードである (一部のメソッドを省略してある)．実験ではこのスタブコードをバイトコードにコンパイルした上で解析に利用される．Collection はインターフェースではあるが，インターフェース共通のデータの振る舞いをするクラスとしてスタブコードを記述することで個々のクラスをもとにしたスタブコードと同様に解析に利用することが可能である．またパッケージ名は，オリジナルのクラスの階層構造を残すため，summary.java.util としてある．

3.1.2 クラスの表現

クライアントサイドから利用されるクラスにはオリジナルのコードと同じクラス名をつける．オリジナルのクラスに存在する継承関係を表現する必要はない．内部の private なクラスについてはクライアントサイドにそのオブジェクトが利用されるのであれば定義する．

スタブコード例では Collection インターフェースをスタブコードとして表現しているので同様に Collection という名前のクラスを定義している．内部クラスとして Itr が存在するが，これは iterator の戻り値として Itr のオブジェクトをクライアントサイドに返すため，定義

```
package summary.java.util;

import java.util.Iterator;

public class Collection<E> {
	public E content;

	public boolean add(E e){
		content=e;
		return true;
	}

	public boolean addAll(java.util.Collection<? extends E> c){
		content=c.iterator().next();
		return true;
	}

	public Iterator<E> iterator(){
		return new Itr();
	}

	private class Itr implements Iterator<E>{

		public boolean hasNext() {
			return false;
		}

		public E next() {
			return content;
		}

	}

	public Object[] toArray(){
		Object[] array={content};
		return array;
	}
}
```

図 4: java.util.Collection インターフェースのスタブコード

している．また Itr は java.util.Iterator インターフェースを継承しているが，これは iterator の戻り値と型を一致させ，コンパイルを通るようにするためである．

3.1.3　変数・フィールドの表現

オリジナルのコードにおいて public や protected なフィールドはスタブコード中でも定義する．private で内部からのみ参照可能な場合や，ローカル変数などは省略する．ただし後述するメソッドの表現においてデータフローを表現するのに必要であれば，オリジナルのコードにないフィールドや変数を定義してもよい．

スタブコード例では元々の Collection に public, protected なフィールドは存在しないため，定義していない．また実際には存在しないフィールドとして content を定義している．これはメソッドのデータフローを表現するために定義されている．詳しくは次項で説明する．

3.1.4　メソッドの表現

public, protected なメソッドのみを実装し，private メソッドは無視する．メソッド名についてはオリジナルのメソッド名と同じものにし，戻り値の型，引数の数や型についてもオリジナルと同じにする．メソッドの実装は引数と戻り値の関係性のみを表現する．引数のオブジェクトが複数の処理を経て戻り値として返るのであれば，スタブコードでは引数を直接 return するのみとなる．あるメソッドが呼ばれたときの引数が，そのメソッドが別の機会で呼ばれた，あるいは別のメソッドが呼ばれたときに戻り値として返る場合は，その引数を保持する private フィールドを定義する．そして引数を与えたメソッドではそのフィールドに引数を代入し，戻り値として返すメソッドではそのフィールドを戻り値として返す．

スタブコード例では add を呼び出したときの引数や addAll の引数のオブジェクトの内部の要素が，toArray の戻り値の配列の要素や Itr#next の戻り値として得られるということを表現するために，content フィールドを定義している．add では content に引数を代入し，addAll では iterator を利用して内部の要素を取り出して content に代入している．Itr#next ではそれを戻り値として返しており，toArray はそれを要素とした配列を戻り値として返している．なお，add や addAll の戻り値，hasNext の定義はコンパイルに通るようにするためのダミーの処理でありデータフロー上の意味はない．

3.2　ポインタ解析への適用

スタブコードを用いてポインタ解析のアルコリズムを適用する際は，解析対象のコードの中にスタブコードも含めて同時に解析を適用する．

そして，解析中にクラスやメソッドを参照するとき，そのコードが存在しなければスタブ

コードが存在するか確認する．クラスの継承などが存在する場合，あるクラスのメソッドの定義がそのクラスに存在しなければ，継承元のクラスを順番に辿っていく場合がある．そのときは，継承元のクラスに辿る前にスタブコードの存在を確認することになる．もし継承元，継承先両方のクラスのスタブコードを用意している場合は，継承先のクラスのオリジナルコード→継承先のクラスのスタブコード→継承元のクラスのオリジナルコード→継承元のクラスのスタブコードの順番で検索する．

例として前述の Collection のスタブコードを用いた Java のポインタ解析において java.util. ArrayList#add の呼び出しが登場した場合を考える．まずは ArrayList#add 自体のコードが解析対象中に存在するかを確認し，次に ArrayList のスタブコード自体やその中の add の定義が存在するかを確認する．その後は java.util.AbstractList クラスや java.util.List インターフェース，java.util.Collection インターフェースといった ArrayList のスーパークラスやスーパーインターフェースについて，1 つずつオリジナルコード→スタブコードの順番で検索しつつ遡る．最終的に Collection インターフェースのスタブコードに add の定義があることを発見し，それを解析するという流れになる．

3.3 スタブコードの利点

Java にはクラスの多態性が存在し，そのため解析中ではある変数のメソッドを呼び出しているときに実際はその変数がどのオブジェクトを指しどのメソッドが実行されるのかを特定する必要がある．解析中にメソッド 1 つを処理するためには，以下の 3 ステップが必要になる．

1. 呼び出し先のメソッドを特定し，呼び出しの引数を呼び出し先のメソッドに伝播させる
2. 呼び出し先の解析を行う
3. 戻り値を呼び出し元に伝播させる

実際の手順では各ステップについて，該当する代入文やメソッド呼び出しの文から有向辺を作成する処理の中で行われている．そのため，プログラム全体をまず 1 ループして呼び出し元から呼び出し先に引数が伝播し，2 ループ目で呼び出し先メソッド内で伝播した情報から新たな辺が生成され，3 ループ目でその戻り値が呼び出し元に伝播するといったケースも考えられる．こういった場合では　1 つのメソッドでプログラム全体の解析を 3 ループすることになる．そのためメソッド呼び出しの深さがより深い場合にはより多くループが必要になり多大なコストがかかる可能性がある．その点スタブコードでは引数と戻り値の関係だけを表現しており，内部でのメソッドの呼び出しが少なく済んでいる．そのため，メソッド呼び出しの点でオリジナルのコードより解析のコストを下げることに成功している．例としてス

タブコードの Collection#add の定義にはメソッド呼び出しが一回も存在しない．一方でオリジナルの ArrayList#add のコードには，いくつかのメソッド呼び出しが存在し，さらにその内部で推移的なメソッド呼び出しが存在する．その結果メソッド呼び出しの深さは 5 以上になっている．そのため前者の方がメソッド呼び出しの解決コストは低く済む．

表 1: 実験に使用したプロジェクト一覧

プロジェクト名	バージョン	クラス数	メソッド数
ANTLR	4.0	745	4889
DrawSWF	1.2.9	322	2840
Jasml	0.10	50	265
JavaCC	5.0	154	2185
JMoney	0.4.4	193	925
JUnit	4.11	233	1305

4 評価実験

4.1 実験方法

対象言語は Java とし，Rountev らの Java 版 Andersen ポインタ解析 [12] 及び，同様のポインタ解析に前述したスタブコードの参照機能を加えたものを用いた．実験対象として，ANTLR, DrawSWF, Jasml, JavaCC, JMoney, JUnit の計 6 つのプロジェクトを用意し，これらに対して 2 つの解析を適用した．各プロジェクトの詳細な情報は表 1 の通りである．解析所要時間の関係上，小規模なプロジェクトを対象として選定しており，これらのプロジェクトはバイナリファイル容量として 100KB から 1.1MB 程度までのプロジェクトである．スタブコードについては java.util パッケージの Collection, List, Set, Map, Queue, Deque インターフェースのものを手書きで用意した．これらのインターフェースは複数のオブジェクトの管理に使われるものであるため，より多くの有向辺の生成に関わっていると思われる．解析を実行したマシンのスペックは CPU が Intel Xeon E5-1650，メモリは 32GB である．

4.2 評価方法

ポインタ解析において変数がオブジェクトを指す辺を後述する内部辺，外部辺に分類する．

内部辺とは，それぞれの変数からその変数が指しうるオブジェクトに対して伸びた有向辺である．なお今回の解析ではプリミティブ型変数に関しては解析の対象としないものとする．

外部辺とは，ポインタグラフが構築出来ないために個別のオブジェクトを指すことが出来ない外部ライブラリのメソッドの戻り値の集合を 1 つの『外部オブジェクト』と定義したとき，その『外部オブジェクト』を指す有向辺である．図 5 のようなコードでは Hoge を含む外部ライブラリを解析対象に含んでいないと，Hoge#get の戻り値が Hoge#set の引数であることが分からない．そのため，Hoge#get 内部の有向辺グラフを必要とする部分では変数 f から正しく本来指すべきオブジェクトへ有向辺を引くことができない．そこで，Hoge#get

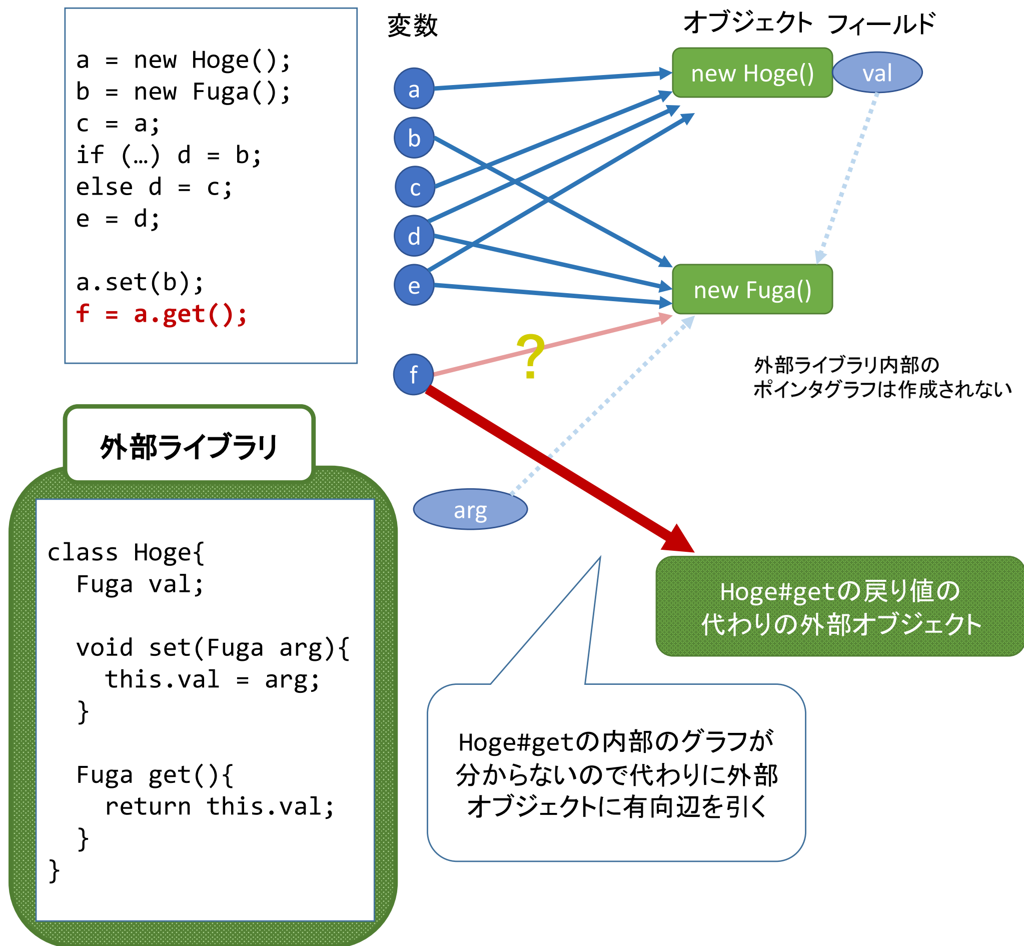

図 5: 外部辺の一例

の戻り値の集合を Hoge#get 由来の『外部オブジェクト』と定義し，1 つのオブジェクトとして扱う．つまりある変数が外部ライブラリの Hoge#get の戻り値になりうるオブジェクトを指すことを，その変数の Hoge#get 由来の『外部オブジェクト』に対して有向辺を引くということで表現する．その外部オブジェクトに伸びた有向辺が外部辺である．なお今回の実験では，外部辺のうち，減少しても解析精度の向上に寄与していないと考えられる一部の辺を計測の対象から除外している．例えば java.lang パッケージの Integer#valueOf の場合は，戻り値が必ずメソッド内で新たに生成されたオブジェクトであり，クライアントプログラムで生成されたオブジェクトである可能性はない．そのため外部辺をスタブコードによって解決しなくても，『Integer#valueOf 由来の外部オブジェクト』はつまりそのメソッド内で生成されたオブジェクトと等価であると言える．一方で java.lang.reflect パッケージの Field#get などは，動的に戻り値や挙動が変わるため，静的解析であるポインタ解析では本質的に対応できない場合などの理由で削減できない外部辺については計測の対象から除外している．外部辺にカウントしないメソッドと理由は表 2 の通りである．

評価実験で適用した解析のそれぞれについて，内部辺，外部辺の個数及び実行時間を計測し，比較する．

表 2: 外部辺としてカウントしないメソッド

除外するメソッド	理由
プリミティブ型のラップクラス (Integer など) のメソッド	ラップしたデータと他オブジェクトの変換を行ったり，計算後の値を返すため，戻り値がメソッド内部で生成されたオブジェクトやプリミティブ型である
java.lang.String のメソッド	文字列の変換メソッドなどが含まれるが戻り値は別の新しいオブジェクトとなる
java.lang.Class のメソッド	動作や戻り値が動的に決定されるもので静的解析では追跡が不可能
java.lang.reflect パッケージのクラスのメソッド	同上
java.io パッケージのクラスのメソッド	戻り値として得られるオブジェクトは外部（プログラム外）からのデータをオブジェクトとして表現したものであり，クライアントプログラム内のオブジェクトである可能性はない
Object#clone メソッド	スタブコードの範囲では正確な扱いが困難
Object#toString メソッド	オブジェクト自身の状態などを表す新たな文字列オブジェクトが生成され，返されることが一般的

内部辺はポインタ解析の解析結果そのものなので，内部辺が増加していれば，ポインタ解析の精度の向上を表していると言える．また外部辺はライブラリ内部の解析ができない結果として生まれるものであり，解析時にスタブコードによって代用した場合には，代用できたメソッドの外部辺は実際に代入されうるオブジェクトに対する有向辺に置き換わる．そのため，減少した外部辺の数や割合を計測することで，作成したスタブコードの効果を測定することができる．

4.3 実験結果

スタブコードの適用の有無ごとの各辺数を表 3 にまとめ，解析時間は表 4 にまとめた． また各プロジェクトごとに，外部辺に関連したメソッドの外部辺数上位 3 メソッドについて，スタブコード適用前は表 5，適用後は表 6 にまとめた．

全てのプロジェクトにおいて内部辺の増加，外部辺の減少，そして解析時間の増大が見られる．内部辺については微増から 1.6 倍程度の増加している．外部辺については 20%から

表 3: スタブコード適用の有無ごとの辺数比較

	外部辺数			内部辺数		
	適用なし	適用あり	前後比率	適用なし	適用あり	前後比率
antlr	985676	225782	22.9%	919744	1259490	136.9%
drawswf	7490	5011	66.9%	32801	40589	123.7%
jasml	108	38	35.1%	1528	1706	111.6%
javacc	29749	6129	20.6%	27688	36603	132.2%
jmoney	563	465	82.5%	4568	4581	100.3%
junit	14491	5269	36.3%	10452	16359	156.5%

表 4: スタブコード適用の有無ごとの解析時間比較

	解析時間 (ms)		
	適用なし	適用あり	前後比率
antlr	5461238	6363944	116.5%
drawswf	2577	4668	181.1%
jasml	1702	1812	106.5%
javacc	4107	12693	309.1%
jmoney	719	844	117.4%
junit	3857	8105	210.1%

80%程度減少している．また，実行時間については微増から最大で 3 倍程度の増加となっている．それぞれプロジェクトごとに差があるが，内部辺，外部辺の増減幅が小さいものほど解析時間の増加が少ない傾向が見られる．

4.4 考察

効果の幅はあるものの，内部辺の増加，外部辺の減少を確認することができた．antlr や javacc と言った，元々の外部辺数の多いものは目覚ましい減少数であることが分かる．外部辺の減少数ほど内部辺が増加していないのは，外部辺の実態として現れた有向辺の指すオブジェクトが，既に存在した内部辺によって指されていた場合などが存在するためだと考えられる．

表 6 の外部辺の内訳を見てみると，Collections クラスや Arrays クラスといった Collection インターフェースに関連したユーティリティクラスや，Stack といった Collection インターフェースを継承したクラスがいまだに多くを占めることが分かる．実験対象としたプロジェ

表 5: スタブコード適用前のプロジェクトごとの外部辺に占める上位 3 メソッド

プロジェクト名	メソッド名	外部辺数
antlr	java.util.List#iterator	540874
	java.util.Collection#iterator	184710
	java.util.List#get	127529
	上位 3 メソッド合計	853113
	プロジェクト全体合計	985676
drawswf	java.util.Vector#get	2022
	java.awt.image.WritableRaster#createWritableChild	1444
	java.awt.image.Raster#createWritableRaster	733
	上位 3 メソッド合計	1794
	プロジェクト全体合計	7490
jasml	java.util.ArrayList#toArray	51
	java.util.ArrayList#get	13
	org.w3c.dom.NodeList#item	13
	上位 3 メソッド合計	77
	プロジェクト全体合計	108
javacc	java.util.List#iterator	17319
	java.util.List#get	9912
	java.util.Map#ketSet	1663
	上位 3 メソッド合計	28894
	プロジェクト全体合計	29749
jmoney	java.util.Vector#elementAt	261
	java.util.AbstractList#listIterator	85
	java.util.Enumeration#nextElement	33
	上位 3 メソッド合計	379
	プロジェクト全体合計	563
junit	java.util.List#iterator	10296
	java.util.ArrayList#iterator	1996
	java.util.Collection#iterator	592
	上位 3 メソッド合計	12884
	プロジェクト全体合計	14491

表 6: スタブコード適用後はプロジェクトごとの外部辺に占める上位 3 メソッド

プロジェクト名	メソッド名	外部辺数
antlr	java.util.Stack#peek	70100
	java.util.Arrays#asList	54434
	java.util.Arrays#copyOf	23365
	上位 3 メソッド合計	147899
	プロジェクト全体合計	225782
drawswf	java.awt.image.WritableRaster#createWritableChild	1945
	java.awt.image.Raster#createWritableRaster	991
	java.util.Enumeration#nextElement	308
	上位 3 メソッド合計	3244
	プロジェクト全体合計	5011
jasml	org.w3c.dom.NodeList#item	13
	org.w3c.dom.Node#getChildNodes	11
	org.w3c.dom.Node#getNodeValue	6
	上位 3 メソッド合計	30
	プロジェクト全体合計	38
javacc	java.util.Enumeration#nextElement	4078
	java.util.Vector#elementAt	2016
	java.util.Hashtable#keys	13
	上位 3 メソッド合計	6107
	プロジェクト全体合計	6129
jmoney	java.util.Vector#elementAt	261
	java.util.Enumeration#nextElement	33
	java.util.Vector#lastElement	32
	上位 3 メソッド合計	326
	プロジェクト全体合計	465
junit	java.util.Arrays#asList	2102
	java.lang.Iterable#iterator	1152
	java.util.Collections#unmodifiableList	742
	上位 3 メソッド合計	3966
	プロジェクト全体合計	5269

クトはバイナリファイルの容量比で高々10 倍程度の規模の近いプロジェクトである．しかし近い規模にも関わらず外部辺数にかなりの差が生まれていることを考えると，規模よりも Collection インターフェースの利用頻度の方が外部辺の数に影響しているように見える．Collection インターフェースはオブジェクトの管理に幅広く使われることから外部辺の定義上，外部辺の数に深く関係していることが予想できたが，実際にその関係性が証明できたと言えよう．

解析時間については元々のポインタ解析から追加の処理を行っているため全てのプロジェクトで増加している．しかし元の解析時間からは大きくとも 3 倍程度の増大であり，新たに多くの内部辺を導出できるようになったことを考慮すると時間の増加幅は許容範囲内に収まっているように思われる．

5　手法の拡張に向けた議論

5.1　スタブコードの追加

今回の実験では試作として一部のインターフェースについてのスタブコードを作成した．作成したスタブコード数は少ないながら現時点で多くの外部辺を削減することができている．ここからさらなる外部辺削減を目指す場合はスタブコードの追加が必要である．もし必要なスタブコード量が膨大なものになるのであれば，実現するにはスタブコード生成の自動化が必要である．自動化をするには，一度は元のライブラリを解析しその情報に基いてスタブコードを生成するプログラムを作成する必要がある．また，その場合でもライブラリに含まれるネイティブコードに関しては解析することができないために別途の対応などを考えるなど一定以上の手間が必要である．

一方で，表 6 の通り外部辺を生み出すメソッドには大きな偏りがある．複数のプロジェクトにおいてその傾向が java.util パッケージ内のメソッドに偏っており，Collection インターフェース，つまりオブジェクト管理をする一部に集中していることが明らかとなっている．またそれ以外のプロジェクトについても外部辺の関係するメソッドは大きく偏っていることが見て取れる．つまりポインタ解析に関しては，一般的なプロジェクトを対象とするとオブジェクト管理に関係しているごく一部のコードのみをスタブコード化すれば十分な効果を得ることができると言える．独自の傾向が見られるプロジェクトについても，少数の偏りの見られる部分についてスタブコードを実装すれば効果を得ることができる．その場合，一つ一つのスタブコードの作成自体はあまり労力を必要としないため，自動化を実装するよりコストが小さくなると考えられる．

5.2　他のデータフロー解析への適用

スタブコードはオリジナルのコードに比べて多くの情報を削っており，それによって解析のパフォーマンスを向上させている．例えば，今回の実験で利用したポインタ解析においてはプリミティブ型について考慮しておらず，スタブコードからもプリミティブ型に関する一切の情報を削っている．

他のデータフロー解析に対してスタブコードを利用する場合，例えばプリミティブ型に関する情報を必要とする解析には今回作成したスタブコードを使うことはできない．他にもプログラムの処理がどの順番で実行されるかを表す制御フローを考慮する解析の場合にもスタブコードの変更が必要である．今回作成したコードでは Collection が保持する要素を単体の変数 content として表現している．そのため仮に add が 2 回連続で呼ばれた場合には，content 変数に 2 回連続で値を代入しているのと同じことになる．制御フローを考慮した解析では，変数に新たなオブジェクトを代入した場合それ以降の処理では以前のオブジェクト

は参照されないという解析が行われる．そのため 1 回目に add されたオブジェクトは無駄になってしまい，add されたオブジェクトは複数同時に Collection 内部に保持されるという本来の Collection の挙動を表すことが出来ていない．これを解消するには，content を配列として表現するなど別のスタブコード表現を考える必要がある．一方で第三者が，ある解析のためにプリミティブ型の情報や制御フローを表現したスタブコードを作成した場合，そのスタブコードは他のプリミティブ型の情報や制御フローを必要とする解析手法でも利用できる．

また，どのような詳細度のスタブコードであっても，スタブコードは形としてはソースコードであるため，今回のポインタ解析のように解析対象に含め，オリジナルの代わりにスタブコードを参照するための処理を追加するという労力の少ない形での利用が可能であると考えられる．例えば 2.3 節で挙げた関連研究で用いられた Taint Analysis についても，外部ライブラリの代わりのスタブコードを直接解析対象に加えることで，データフローを網羅する複雑なサマリーの表現の考案や，そのサマリーを作成するプログラムを作成する必要がなく解析が行える可能性がある．その点ではスタブコードはデータ依存解析の中間フォーマットとして有用であると言える．

一方で，適用する解析手法において必要なデータフロー情報が増えるほど，スタブコードの作成は煩雑になっていく．しかし関連研究では個別の研究ごとに外部ライブラリから必要な情報を集めてサマリー化していたため，サマリーのフォーマットとしてスタブコードを採用することは既存の研究からそれほど手間が増えることはないように思われる．また，スタブコードに表現すべきデータフロー情報が多い場合では，今回の実験において見られたような傾向 (Collection インターフェースに関連するなど) が希薄になり，よりスタブコードの作成が必要な範囲が拡大して必要な作業量が増加すると考えられる．その場合は前述した自動化も費用対効果という観点から有効となってくる可能性がある．

6　まとめ

本研究ではポインタ解析において，外部ライブラリの解析に必要なコストを削減するための手法として，スタブコードという外部ライブラリの表現方法を提案した．スタブコードは解析にそのまま含めることができるため扱いやすいという利点がある．その手法について評価実験を行い，ライブラリを完全に無視する場合と比べて精度の高い解析を行えることを確かめた．また，外部ライブラリの中で解析の精度に影響が出やすい部分が偏っていることも明らかになり，スタブコードの作成により効率良く精度の改善が行える可能性を示すことができた．

今後の課題としては，多様な外部ライブラリへの有効性の調査が挙げられる．実験では外部ライブラリとして Java クラスライブラリに焦点を当てている．これは Java の標準ライブラリとも言えるもので，どのようなプログラムでも利用されうる一般的な機能が実装されているものである．したがってそれは各プログラムの目的に基づいた機能とはあまり関係がなく，より特殊な機能が実装されたライブラリに対して実験を行えば異なる外部辺の傾向を示す可能性がある．そのため標準ライブラリだけでなく様々な用途のライブラリを使用したプロジェクトに対する外部辺の傾向を調査し，スタブコードの利用が有効である条件を調査することが必要であると言える．

謝辞

本研究において, 常に適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に心より深く感謝いたします.

本研究において, 適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠准教授に深く感謝いたします.

本研究において, 逐次適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻石尾隆助教に深く感謝いたします.

本研究において, 様々な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻竹之内啓太氏に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします．

参考文献

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Department of Computer Science, University of Copenhagen, 1994.

[2] Steven Arzt and Eric Bodden. Stubdroid: Automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering*, pp. 725–735, 2016.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices*, Vol. 49, No. 6, pp. 259–269, 2014.

[4] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 576–587, 2014.

[5] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications. Technical report, Department of Computer Science, University of Maryland, College Park, 2009.

[6] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information flow analysis of android applications in droidsafe. In *he Network and Distributed System Security Symposium*, 2015.

[7] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 106–117, 2015.

[8] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In *Mobile Security Technologies*, 2012.

[9] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a certifying app store for android. In *Pro-*

ceedings of the 4th ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 93–104, 2014.

[10] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pp. 229–240, 2012.

[11] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.

[12] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. *ACM SIGPLAN Notices*, Vol. 36, No. 11, pp. 43–55, 2001.

[13] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *In Proceedings of the 21th Annual Network and Distributed System Security Symposium*, 2014.