Method of Inferring Source Code Locations Corresponding to Mobile Applications Run-time Logs

博　士　論　文

実行時ログから対応する
ソースコード箇所を推測するモバイル
アプリケーションの開発支援手法

金沢大学大学院自然科学研究科
電子情報科学専攻

学籍番号　：　１３２３１１２００２
氏　　名　：　小野　祐貴
主任指導教員名　：　山根　智
提出年月日　：　平成28年1月

目次

第 1 章　はじめに

近年, スマートフォンやタブレットなどで動作するモバイル OS のアプリケーション (以下, モバイルアプリケーションまたは単にアプリケーションと呼ぶ) の開発が盛んである. モバイル OS は代表的なものに Google 社の Android OS や, Apple 社の iOS, Microsoft 社の Windows Phone などがある. モバイルアプリケーションの開発は, 多くの場合開発環境と実行環境が異なり, 開発環境でソースコードを記述し, スマートフォンやタブレットなど端末上の実行環境で実際に動作確認を行う. モバイルアプリケーションの動作保証のためには, 開発者は存在する複数の実行環境に対応したソースコードを作成し, それらの実行環境で動作確認とデバッグを行う必要がある.

一般的に開発者はモバイルアプリケーションの実行時の動作が期待どおりか確認するために, アプリケーションのソースコードに実行時ログを生成するためのログ関数呼出しを埋め込む. 多くの場合, 実行時ログは開発者に対してヒントとなる文字列定数や実行時の変数の値を含むテキスト列であり, ソースコードに挿入されたログ出力用の手続き呼出しから生成される. アプリケーションがプログラマの意図しない動作を含む場合は, 意図しない実行時ログが得られるようにすることができる.

意図しない動作の箇所を特定するためには, 生成された実行時ログからプログラムの実行時の動作を詳細に推定する作業が必要になる. 特にモバイルアプリケーションの動作する環境は一般的にリソースが限られているので, 実行履歴を記録するデバッガ [8] などは適用が難しいため, 実行時ログが頼りになる. さらに多様な実行環境で生じる意図しない動作を再現させ, 実行時ログから関連する部分を特定することは簡単ではなく, デバッグ作業の時間増加につながっている. 例えば, Android OS では, カメラ撮影を行うアプリケーションの動作が端末によって異なるため, 特定の端末の意図しない動作の原因を実行時ログから推定することになる (具体例として第 2 章節で詳細に説明する.).

本研究では, 実行時ログとソースコードの対応を行い, 実行時ログを生成した箇所の候補を推測することで, 開発者の実行時の動作の推定作業を支援するツール *LogChamber* の提案を行う. LogChamber は対象アプリケーションのプログラムをあらかじめ解析し, ログ出力を行うログ関数呼出しの箇所を特定する. さらに, ログ関数呼び出しの引数を詳細に解析することで, 実行時ログと高速に対応付けを可能とする索引を作成する. LogChamber によって開発者は場当たり的に挿入された実行時ログの箇所を特定し, アプリケーションの動作を分析することが容易になる. また, LogChamber の推測と表示は高

速なので, 動作中の端末に接続し, アプリケーションがログ出力を生成した時点で対応するソースコード上の箇所を特定し表示することができる.

本研究では, LogChamber を Android OS のアプリケーションの統合開発環境 (Android Studio) のプラグインとして実装し, 実際の Android アプリケーションに対して適用する実験を行った. 本稿では評価のために, 実際のアプリケーションに存在するプログラマの意図しない動作の箇所を, LogChamber を使って特定する手順を示す.

第 2 章節では, モバイルアプリケーションのデバッグでの問題を具体的な例とともに説明する. 第 3 章節では LogChamber を提案し, そのアルゴリズムについて詳細に説明する. 第 4 章節では実際のオープンソースアプリケーションに対して LogChamber を適用した評価実験について説明する. 第 5 章節では関連研究について述べる. 第 6 章節はまとめである.

第 2 章　モバイルアプリケーションのデバッグ

ここではモバイルアプリケーションの動作確認とデバッグの問題を具体例により説明する.

モバイルアプリケーションは, デスクトップアプリケーションや Web アプリケーションとは異なり, リソースが少ない環境で動作し, 端末の種類が多く, それらのハードウェア構成が多様なため, 特定の端末に依存した動作と関連する意図しない動作が生じやすい. 以降, 2.1 節では具体的な例を挙げ, モバイルアプリケーションの実行時ログを使った意図しない動作のデバッグ手順を説明する. 2.2 節では一般的なブレークポイントデバッガとの比較により, モバイルアプリケーションのデバッグの問題点を述べる. 2.3 節では実行時ログをつかった場合の問題点をまとめる.

2.1　例: カメラアプリケーションの傾き方向の取得

図 1 は Android アプリケーションのカメラ撮影の機能を実装するコードの一部である. このコードは, 特定の端末と環境に対する意図しない動作があり, 撮影した画像がユーザにとって不自然な角度で回転して表示する意図しない動作を行う. 意図しない動作を確認し修正するため, デバッグのログ出力を行うコードを挿入している (4,7,13,15,18 行目).

ユーザがカメラで撮影を行うと, コールバックとして `onPictureTaken` メソッドが呼出され, 撮影した画像の JPEG データが引数 `data` に与えられる (2 行目). まずこの画像をファイルに保存し (4–9 行目), Exif データ (メタデータ) から撮影時の端末の傾き方向を取得して `r` に代入する (10 行目).

意図しない動作の修正として, 端末でカメラ撮影を行い撮影した画像の JPEG データを表示する際に, 必要に応じて画像データと回転方向を取得し, 適切な画像の向きを決定する必要がある. 例えば, 撮影時に端末を 90 度回転させた状態で撮影した際に, 端末により生成される JPEG データは 90 度回転しており, そのままの向きで画像を表示すると, 利用者から見て不自然に回転した画像を表示されることになる. そのため, JPEG データのメタデータの値を取得して確認し, 実行時に手動で適切に回転し表示する必要がある.

13,15 行目はデバッグ用のログ出力のためのログ関数で, r の値をメッセージとともに出力する.

例えば “`JPEG rotation = 1`” の場合は JPEG データをそのまま表示するという意味

であり, アプリケーションで画像のデータは回転する必要はない. r は 1–8 の値があり[*1], 画像の表示の際にはこれらの値に対応した回転処理をする必要がある. さらに r = 0 (UNDEFINED) となる端末があり, その場合は画像の縦横比などから特別な回転処理を行う必要がある[*2].

2.2 ブレークポイントデバッガの利用と問題

アプリケーションの動作確認とデバッグを行うためのツールとしてブレークポイントデバッガが広く普及している. ブレークポイントデバッガではプログラムの任意の箇所にブレークポイントを設置して実行することで, その箇所で実行を一時的に停止させ, 変数の値などの状態を確認できる. また停止後にステップ実行をすることでアプリケーションが想定した動作を行うかを確認できる. 例えば, 図 1 のコードに対しては, 11 行目にブレークポイントを設定し実行することで変数 r の実際の値を確認することができる.

しかし, ブレークポイントデバッガではブレークポイントを管理したり, ステップ実行を行うための操作が多くなり, デバッグのための作業量が大きくなることがある.

また, デバッグのためには意図しない動作を再現させる必要がある. 2.1 節で示したような一部の端末や環境で生じる意図しない動作を, ブレークポイントデバッガで再現させる作業は現実的でない. 意図しない動作が再現する端末が開発者の手元になく, ユーザが送信したログ出力を分析しなければならない場合もある.

結果として開発者はアプリケーション内でログ関数を使用し, 実行時ログと実際のソースコードを照らし合わせて, 実際のプログラムの動作を推定することになる. そのような推定は開発者の能力に依存し, 一般的に簡単ではなく, デバッグの作業時間の増加につながる.

2.3 実行時ログの一意性と特定の問題

一般に, 実行時ログは他のアプリケーションや OS によるログ出力を含むため, 開発者はその中からデバッグ対象のアプリケーションのログ出力を特定しなければならない. アプリケーションの特定ができたら, 次にアプリケーション内のソースコード箇所の特定が必要になる.

図 2 は図 1 のコードを含むアプリケーションの実際の実行時ログである. このログ

[*1] http://www.sno.phy.queensu.ca/~phil/exiftool/TagNames/EXIF.html

[*2] http://hackmylife.net/archives/7400448.html

```
1  // JPEG イメージ生成後に呼ばれるコールバック
2  public void onPictureTaken(byte[] data,
3                             Camera camera) {
4    Log.d(TAG, "Start Save JPEG");
5    boolean failed = saveImageToFile(data,FILE_NAME);
6    if (failed) {
7      Log.d(TAG, "Save JPEG Failed");
8      return;
9    }
10   int r = getExitInterface(FILE_NAME);
11   if (r != 0) {
12     //JPEG の回転の値が端末によって異なる
13     Log.d(TAG, "JPEG rotation = " + r);
14   } else {
15     Log.d(TAG, "Illegal rotation = " + r);
16     return;
17   }
18   Log.d(TAG, "Saved JPEG Size " + data.length);
19   camera.startPreview();// プレビューを再開する
20 }
```

図 1　端末によって結果が異なる写真撮影コードの例

は特定するためのヒントとして, 実行時間, プロセス ID, タグ文字列が含まれる. タグ文字列はログ関数に与える文字列定数として利用され, 図 1 の例では 13 行目の第 1 引数の TAG が相当する. 開発者は 13 行目の TAG にあらかじめ"screen2camera"という文字列を設定することでログ出力の特定に役立てることができる. 図 2 では 1 行目は "screen2camera"を含まないので, API によるログ行であることがわかり, 2,3 行目が含むので図 1 の 13 行目から生成されたログ行だとわかる.

しかし, 必ずしもログ関数の呼出しが計画的に特定できるように書かれるわけではなく, ログ出力の一意性は保証されない. 例えば, カメラ撮影の API を使う箇所が複数あれば, 開発者はそこに 13 行目と同様のログ関数呼出しを書く可能性がある. 実際に図 1 の 4,15,18 行目では同じタグ文字列 TAG を利用している.

```
1  10-04 04:31:31.076 820-5820/camera_sample
     W/Resources Converting to string:
     TypedValue{t=0x5/d=0xa01 a=1 r=0x10500d8 }
2  10-04 04:34:34.887 820-5820/camera_sample
     D/screen2camera Start Save JPEG
3  10-04 04:34:35.023 820-5820/camera_sample
     D/screen2camera Illegal rotation = 0
```

図 2　実際の実行時ログの例

ログ出力はデバッグのために, 繰り返し挿入や削除を行うことで, 結果として場当たり的に挿入されがちである. ログ関数呼出しの書き換えを頻繁に行うため, TAG のような特定のヒントであっても, Java のパッケージやクラス単位といったある程度の大きさの粒度で設定され, (具体的なソースコード行ではなく) 対象のアプリケーションのモジュールの特定に限定されることが多い.

さらに, ログ出力はソースコード上の式から生成されるため, ログ関数呼出しの式が複雑であれば開発者によるログ出力の特定が難しくなる. 例えば図 1 の 13 行目の `+ r` は文字列 `r` を連結するが, 実際には整数値の文字列であることを開発者が知らなければ具体的なログ出力の特定が難しくなる. この例では単純な式だが, 一般には複数の変数を組合せたログ関数呼出しが複数存在することで, 実行時ログからソースコードの箇所の特定が難しくなり時間のかかる工程になる. いくつかのアプリケーションのログ出力を調べたところ, 英単語で分割できるようなわかりやすいログで出力されていることから, 単語ごとに分割すると, ログ行からソースコード上のログ出力関数を推測できると考えたのが本研究のアイデアである.

第 3 章　LogChamber: 実行時ログから対応するソースコード上の箇所を推測するツール

本研究ではモバイルアプリケーションの実行時ログから対応するソースコードの箇所を推測するツールを提案する．実際に Android アプリケーションを対象とし, 統合開発環境である Android Studio (IntelliJ IDEA) に対するプラグイン *LogChamber* として開発を行った．ここでは, 本研究の提案するツールの手法を, LogChamber とその対象である Android アプリケーションに基づいて説明する．

3.1　概要

LogChamber はアプリケーションの実行時ログとソースコードを解析することで, その実行時ログが発生したソースコードの箇所を推測する．

一般に, 2.3 節で説明したように, 実行時ログには複数のアプリケーションや OS による出力が混在している．またログ出力には容量制限もあり, その情報量は限られている．実行時ログから実際のログを出力した箇所を, 一意に特定することは困難であるため, ログ出力箇所の推測を行う．

図 3 は Android Studio 上で動作する LogChamber の実行例のスクリーンショットである．LogChamber は実行時ログを読込んで, 実行時ログとそのログから推測したソースコード上の断片を画面下部に表示する．

ログ出力箇所の推測は下記の手順で行う．図 4 は LogChamber 全体の構成図で, 各手順に対応した部分の番号を示している．

1. アプリケーションパッケージに含まれるバイトコードから予め登録したログ関数呼出しを抽出する．
2. ログ関数呼出し式の引数を静的に解析し, 引数に含まれる文字列定数を単語ごとに分割する．ログ関数呼出し式の単語リストを作成し, 出現するソースコード上の箇所とともに保持する．
3. 実行時ログを解析して, 一行ごとにスペースなどの特定の区切り文字で分割する．実行時ログの一行ごとの単語リストから, ログ関数呼出しの単語リストに対する一致度を計算する．
4. 最も一致度が高いログ関数呼出しのソースコードの箇所を実行時ログに対するログ

関数の推測候補として表示する. 表示する際には, ログ関数呼出しの引数の値と文字列を対応づけて表示する.

LogChamber を使ったログ箇所の特定は, 以下のような利点があり, モバイルアプリケーションのデバッグの問題を解決する.

- システムが推測を行うため, 開発者がタグ文字列などを厳密に管理する必要がなく, 場当たり的なログ関数呼出しの変更があるコードでも特定が容易になる.
- 保存された実行時ログを入力として与えることで, ユーザーが送信したようなログ出力の動作を擬似的に再現することができる.
- 実行中のアプリケーションのログ出力をその場で入力として与えて推測することも可能で, ブレークポイントデバッガとも連携させることができる.

以降の節では, 各手順について順番に説明していく.

3.2 ログ関数呼出し式の抽出

LogChamber は推測の対象箇所となるログ呼出し式を抽出するため, Android のアプリケーションパッケージ (APK) ファイル内にある Dex ファイルのバイトコードを解析する. LogChamber を実行すると, まず自動的に Android Studio 上でビルドを行い, デバッグ用の情報を含む APK ファイルを生成する. 次に, Java バイトコード解析フレームワークである Soot [11][*3]を利用して解析を行う.

一般的な Java コンパイラが生成するバイトコードは, ソースコードと対応する命令列をほぼそのまま保持するため, メソッド呼出しの抽出のための探索や 3.3 節で説明するデータフローの解析は, ソースコード上で解析した場合と同様の結果を得ることができる.

ログ関数呼出し式の抽出では, バイトコードの解析による探索を行い, 表 1 に示すシグニチャを持つ特定のクラスのメソッド呼出しを, ログ関数呼出し式として見つけ出す. これらは一般的に利用される Android のログ出力関数であり, 開発者は正規表現のパターンによって他のメソッドを指定することもできる.

発見したログ関数呼出し式は, ログ出力箇所の推測のために, 3.3 節で説明する解析の対象となる.

[*3] `https://github.com/Sable/soot` : Dex ファイルの解析に対応したバージョン (commit ID: `36f9765`) を使う.

表 1　Android のログ関数の例

`Log`	Android 標準のデバッグメッセージクラス
`Logger`	Java 標準のデバッグメッセージクラス
`PrintStream#print`	Java 標準の出力関数
`PrintWriter#print`	Java 標準の出力関数
`Console`	Java 標準のコンソールクラス

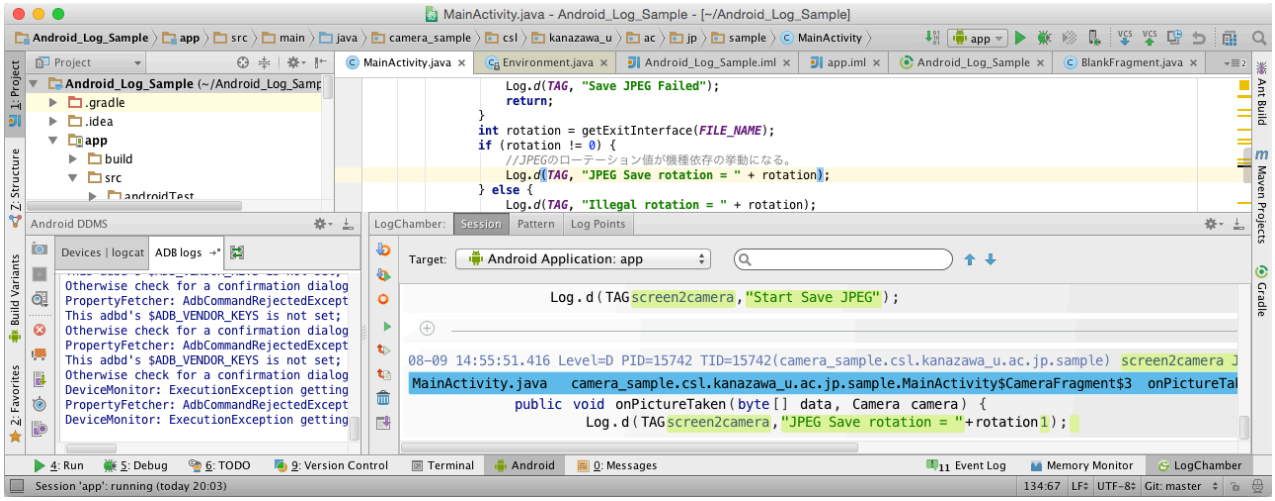

図 3　統合開発環境 Android Studio 上での LogChamber 実行画面

3.3　ログ関数呼出し式の解析

実行時ログから対象のログ関数を特定するために, キーワードとして, 関数の引数に使われている文字列を解析し, 単語リストを作成し, そのログのプログラム上の箇所とともに保持する. プログラム上の箇所と引数の文字列の解析は,　図 5 に示す getLocAndWords 関数を用いる.

getLocAndWords は, x 個の引数を取るログ関数のプログラム表現の構文木上の式 $m(e_1, \ldots, e_x)_{loc}$ を渡すことで, ログ関数のソースコード上の箇所 loc とログとして出力する文字列の単語リスト (w_i) を得る関数である. $\oplus$ は 2 つの単語リストを連結する操作である. $words$ は式から単語リストを得る関数である.

words(e_i) はプログラム中の i 番目の式 e_i の種類に応じて結果を返す.

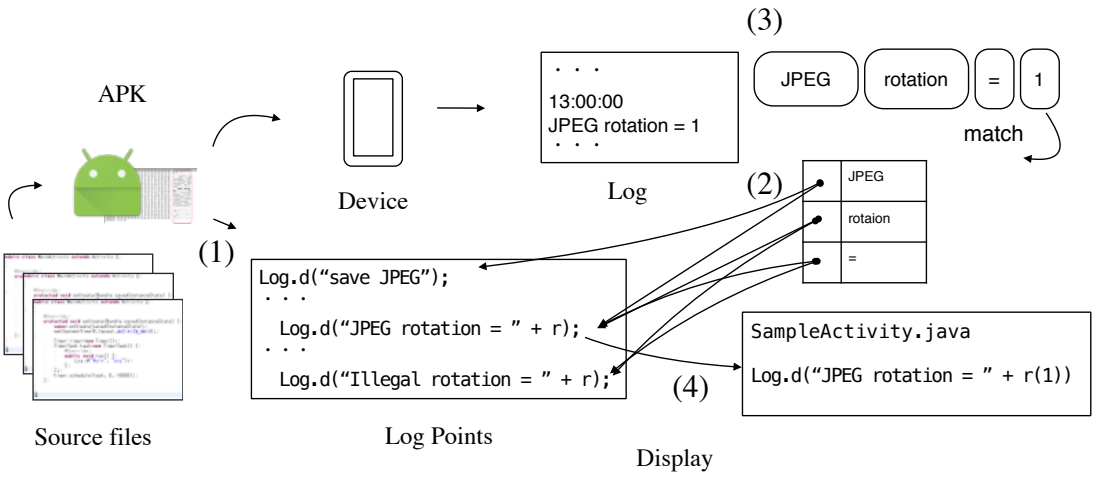

図 4　LogChamber のシステム構成図

- 文字列定数 $Const$ の場合は, 文字列を単語リストに分割する. なおここでの単語は記号類も含む. これは一般にログ出力の文字列は短く, 記号によって特徴付けられるためである. 例えば"JPEG rotation ="という文字列定数からは (JPEG, rotation, =) という単語リストが得られる. なお図 1 の TAG のような文字列の静的フィールドは文字列定数として扱う.
- ローカル変数参照 Var の場合は, 単一に決定される代入式 def(Var) の右側の式 .rhs から再帰的に単語リストを得る. ローカル変数の代入式は分岐の合流などによって複数存在する可能性があるが, ここでは簡単のためにそのような場合を考えない. [*4] このローカル変数の参照式 Var とその代入式 def(Var) を介した words の再帰的な適用は, メソッド内の他のステートメントの式を参照し, 探索することを意味する. このようなローカル変数の展開を行うため, あらかじめメソッド内のデータフローの解析が行われる.
- 文字列の連結式 $e_l + e_r$ の場合は, 左右の式から得られた単語リストを連結する. 実際には解析はバイトコード上で行われ, また Java の文字列連結操作は StringBuilder で実装されるため, この式は実際には StringBuilder クラスの append メソッド呼出しになる.
- ログ出力はフォーマット関数呼出し (String.format) から生成されることが多いため, そのような呼出しを特別に解析する. フォーマット関数は引数 ("c_f%p_f", e_f)

[*4] 実際には, 式全体のローカル変数の代入文の組み合わせごとに個別のログ関数呼出しとして扱う.

$$\text{getLocAndWords}(m(e_1, \ldots, e_x)_{loc}) = (loc, \text{words}(e_1) \oplus \text{words}(e_2) \oplus \cdots \oplus \text{words}(e_x))$$

$$\begin{aligned}
\text{words}(e_i) = \text{match } e_i \text{ with} \\
Const \rightarrow &\quad \text{splitWords}(Const) \\
Var \rightarrow &\quad \text{words}(\text{def}(Var).\text{rhs}) \\
e_l + e_r \rightarrow &\quad \text{words}(e_l) \oplus \text{words}(e_r) \\
\texttt{String.format}(\texttt{"}c_f\texttt{\%}p_f\texttt{"}, e_f) \rightarrow &\quad \text{words}(\texttt{"}c_f\texttt{"}) \oplus (\texttt{\%}f) \\
\text{otherwise} \rightarrow &\quad (\texttt{\%}i)
\end{aligned}$$

図 5　ログ関数の箇所と単語リストを得るアルゴリズム

が与えられる. これらの引数は複数の変換からなり, その f 番目は, 変換の間に連結される文字列 c_f, 変換の指定 $\texttt{\%}p_f$, そして変換の値を生成する式 e_f に対応する. このようなフォーマット関数呼出しに対して, c_f の単語リストと, 単語が変換の式 e_f から生成されたことを示す特別な単語 $\texttt{\%}f$ を連結した列を返す.

- その他の式は式 e_i から動的に生成されることを示す特別な単語 $\texttt{\%}i$ として扱う. なお, これらの特別な単語はのちの表示のために利用し, 実行時ログとの一致度の計算では無視される.

例えば図 1 のソースコードの 13 行目のログ関数呼出しからは,
(`screen2camera`, `JPEG`, `rotation`, `=`, $\texttt{\%}i$) という単語リストが得られる.

3.4　実行時ログの解析と一致度の計算

ソースコードを解析して得られたログ関数呼出しとログ出力の各行を対応づけるため, LogChamber は一致度を計算する. 一致度の計算は, ソースコードのログ関数それぞれから作成した単語リストから, その単語が含まれる全行数分との割合値で表す. 一致度の定義には, 一般に各単語リストの長さは短く, 実行時ログの単語リストには変数から生成されたソースコードの単語リストには含まれない単語が出現することを考慮する.

単語リスト $(n_{i_1}, \ldots, n_{i_{|r|}})$ からなるログ行 r に対して, 単語リスト $(n_{j_1}, \ldots, n_{j_{|l|}})$ からなる各ログ関数呼出し l の一致度は次の式から得られる.

$$s(r, l) = \sum_{k=1}^{|r|} f_1(l, i_k) + f_2(l, i_k, i_{k+1})$$

$$f_1(l,x) = \begin{cases} \frac{1}{|\{l' : n_x \in l'\}|} & \text{if } n_x \in l \\ 0 & \text{otherwise} \end{cases}$$

$$f_2(l,x,y) = \begin{cases} \frac{1}{|\{l' : (n_x,n_y) \lhd l'\}|} & \text{if } (n_x,n_y) \lhd l \\ 0 & \text{otherwise} \end{cases}$$

ここで, $\lhd$ は各ログ関数呼出しの単語リスト l の列で、(n_x, n_y) の順番で連続に含まれていることを表す演算子であるとする. つまり単語リスト中に $(\ldots, n_x, n_y, \cdots)$ のように含まれることを意味する.

f_1 および f_2 は単語の出現頻度を計算する. $|\{l' : n_x \in l'\}|$ は単語 n_x を含む全ログ関数呼出し数である. 同様に $|\{l' : (n_x, n_y) \lhd l'\}|$ は 2 つの連続した単語 n_x, n_y を同じ順序で含む全ログ関数呼出し数である.

すべてのログ関数呼出しに対して一致度 $s(r,l)$ が最大になる l を, ログ行を生成したソースコードの箇所として選択する.

例として図 1 の 5 箇所のログ関数呼出し (4,7,13,15,18 行目) に対して, 図 2 の 2 行目のログ行の対応付けを考える. なお, ここでは簡単のため, 他にログ関数呼出しはないものとする.

図 2 の x 行目の各ログ関数呼び出しの位置を L_x とすると, ログ行の単語リスト (`screen2camera`, `Start`, `Save`, `JPEG`) に対して, 単語それぞれが出現するログ関数呼出しを以下のように求めることができる.

$$(\{L_4, L_7, L_{13}, L_{15}, L_{18}\}, \{L_4\}, \{L_4, L_7\}, \{L_4, L_7, L_{13}, L_{18}\})$$

これらの単語と出現頻度から, 各ログ関数呼出し箇所に対して f_1, f_2 の合計及び s の値は以下の表のようになる.

L_x	$\sum f_1$	$\sum f_2$	s
L_4	$0.2+1+0.5+0.25$	$1+1+0.5$	4.45
L_7	$0.2+0+0.5+0.25$	$0+0+0.5$	1.45
L_{13}	$0.2+0+0+0.25$	$0+0+0$	0.45
L_{15}	$0.2+0+0+0$	$0+0+0$	0.25
L_{18}	$0.2+0+0+0.25$	$0+0+0$	0.45

L_4 に対する f_1 と f_2 の値はすべての単語が出現するため, 0 より大きい値が合計され, s の値が他の箇所よりも大きくなる. 結果として L_4 が最も高い s の値をとる推測候補として, ログ行と対応づけられる.

図 6　ログ関数呼出しとその周辺のソースコード

3.5　ログ関数の推測候補の表示

LogChamber は実行時ログ行に対して, 最も一致度が高い単語リストを持つログ関数呼出しを推測候補として画面に表示する. このとき, 開発者の直感的な理解を助けるため, ログ関数呼出しの周辺のソースコード行と, そのソースコード行の式から生成されたと考えられるログ行の部分文字列をソースコード上に挿入して表示する.

図 6 は表示結果の例である. この図の例のように, 最初の行は出力ログ行を表し, 次の行は対応ソースファイル箇所とそのコード行番号を示す. 3 行目は対応箇所のメソッドシグニチャ部分で, 最後の行がログ関数呼出しをログ出力の部分文字列と組み合わせた表示である.

実行時ログの単語リスト $(\ldots, n_i, n_{i+1}, \ldots, n_{j-1}, n_j \ldots)$ と, 式 e_x から生成される単語を示す %x を含む, 最も一致度が高いログ関数呼出しの単語リスト $(\ldots, n_i, \%x, n_j, \ldots)$ が得られたとする. 実行時ログの部分単語リスト $(n_{i+1}, \ldots, n_{j-1})$ は, %x に対応し, 式 e_x から生成された文字列である. 従って, ソースコード行の表示に際し, 式 e_x の直後に単語リスト $(n_{i+1}, \ldots, n_{j-1})$ を挿入する.

式 e_x に対応する単語リスト上の %x は, 3.3 節の関数 words により得られる. words はローカル変数のデータフロー解析を使った探索が行われるため, ログ関数呼出しを含むメソッド内の式が含まれる. 探索は実際にはバイトコード上で行われるため, 実行時ログの部分文字列の挿入箇所は, 式 e_x に対応するデバッグ用の情報であるソースコードの位置情報を参照して決定する.

第 4 章　評価実験

本研究の提案する LogChamber の有効性を確認するため, 実際にオープンソースで公開されている Android アプリケーションに対して適用する実験を行った.

実験に用いた環境は, LogChamber を実行する開発環境が Mac OS X (10.10.5) , Intel Core i5 1.4GHz CPU, 16GB RAM Java 1.8.0_20, Android Studio 1.3.2 (-Xmx 8g) であり, 対象アプリケーションを動作させる端末が Google Nexus 5 である.

実験では, LogChamber のプログラム解析の性能に関する計測 (4.1 節) と, 実際のアプリケーションの意図しない動作を修正に利用した場合の有効性 (4.2 節) を確認した.

4.1　LogChamber の適用に関する性能

LogChamber はアプリケーションの解析を行うため, 適用時 (3.1 節の手順 (1) および (2)) に開発環境で計算時間やメモリを要する. Android Studio 上で開発プロジェクトをオープンし, ビルドによって生成された APK ファイルに対して LogChamber の解析を適用する際の性能を計測し, LogChamber が現実的に利用できるかを確認した.

また, 計測したソースコードのログ関数呼出しを詳細に分析し, 既存のテキスト検索手法との比較 (4.1.3 節), 提案手法の限界 (4.1.4 節), 及びログ出力関数が互いに成功率に影響を与えるか (4.1.5 節) について議論する.

4.1.1　計測対象と方法

計測対象は Android のオープンソースのアプリを配布するサイトである F-Droid[*5]に登録されているアプリから選択した. これらのアプリは F-Droid の登録リポジトリから自動でビルドし, 生成された APK ファイルに対して実行したもののうち, クラス数が多かった上位 9 個と 4.2 節の実験で使用する Lightning Browser からなる.

LogChamber の性能として, 解析にかかる時間だけでなく, ログ関数の推測候補の精度 (成功率) が考えられるが, 実際の厳密な精度はアプリの動的なログ出力に影響されるため, 定義と網羅的な計測が困難である. そのため今回は, 各ログ関数呼出しから, 3.3 節で説明したアルゴリズムによって静的に抽出される単語リストをログ出力とみなし, それがそのまま抽出したログ関数呼出しが結果の推測候補になるかどうかを調べた.

[*5] `https://f-droid.org`

4.1.2 計測結果

結果は表 2 にまとめてある. 表の各列は以下の値を示している:

App : アプリケーション名.

Time : 解析にかかった時間 (秒).

Classes : APK ファイルに含まれるクラス数. なお APK ファイルはアプリケーションが依存するライブラリなどのクラスを含む (Android SDK は除く).

Methods : APK ファイルに含まれるメソッド数.

LPs : ログ関数呼出し数.

SR : ログ関数の推測候補の成功率. 各ログ関数呼出しから静的に抽出された単語リストをログ出力とみなして, 抽出元のログ関数呼出しが推測候補となった場合に成功とする. 実際には変数出力による動的な値を含んだログ出力が入力となるため, この値より実際の成功率は低下する場合もある.

Words : アプリケーションに含まれる全単語の集合のサイズ.

$|\mathrm{LP}|_{avr}$: ログ関数呼出しの平均単語数.

結果から, 4000 個以上のクラスを持つアプリであっても 100 秒以内に解析が終わっていることがわかる. LogChamber の解析はプログラム全体のソースコードを探索するが, いわゆる手続き内 (intra-procedural) 解析であり, メソッド間の呼出しフローや大域的なデータフローを扱わないため, 高速で十分に実用的と言える.

ログ関数の推測の成功率 (SR) は, 概ね 60–80% となっている. 実際には動的な値から生成された文字列がログ出力に追加されるため, 多くの場合この成功率は低下すると思われる. 一方で全体の値から, アプリケーションのサイズにはあまり関係がなく, ログ関数呼出しの書き方に影響を受けると予想される. 開発者がログ関数呼出しを書く際に, 文字列定数に使用する単語の多様さを意識することで, LogChamber は実用的に動作すると期待できる.

4.1.3 テキスト検索の手法との比較

ここでは本研究の提案する LogChamber と, grep に代表される単純なパターンマッチによるテキスト検索ツールを使った場合との比較を, 計測対象アプリケーションの具体的な例を用いて行う.

ログ出力関数に文字列定数が記述されている場合は, テキスト検索ツールを用い, ある程度のログ出力に対応するソースコード箇所の候補を推測することができる. 例えば grep

表 2　アプリケーション解析の性能

App	Time(sec)	Classes	Methods	LPs	SR	Words	$\|LP\|_{avr}$
Chapel Hill Transit	67.22	4,445	30,481	683	0.61	2,907	6.96
Tacere	84.45	4,433	35,562	454	0.80	2,727	8.79
WiGLE Wifi Wardriving	65.89	3,993	28,230	732	0.62	2,726	6.77
Aizōban	46.53	3,032	21,988	449	0.75	2,264	8.03
Anthology for Gives Me Hope	26.07	2,635	19,329	283	0.81	1,765	9.80
ownCloud News Reader	28.48	2,261	20,257	500	0.83	2,809	8.44
Password Store	55.84	4,363	35,707	358	0.82	2,156	9.59
HackWinds	18.78	1,900	14,187	309	0.81	1,626	8.57
weiciyuan	23.99	1,728	11,902	184	0.93	1,268	10.57
Lightning Browser	23.42	969	6,357	234	0.80	1,356	9.85
平均	44.07	2,976	22,400	419	0.78	2,160	8.74

を使うことで, ログ出力行の任意の単語をワイルドカード (.∗) に置換えてソースコード全体から検索する手法が考えられる.

しかし, 対応するログ出力関数呼出し式が動的な計算によって出力文字列を組立てている場合, 単純なテキスト検索のパターンマッチではうまくいかなくなる.

例えば ownCloud News Reader アプリの `DaoMaster` クラスには図 7 のようなログ関数呼出し箇所がある. ログ出力に含まれる `SCHEMA_VERSION` の値 `4` を検索パターンに含めてしまうと, 単純なテキスト検索では発見できなくなる. このように一般的にはログ関数呼出し式の引数は計算によって合成される場合がある.

まず, テキスト検索のためのパターンはログ出力行から利用者が推測することになるが, 実際には変数から生成される部分文字列と文字列定数による文字列の境界がはっきりしない場合がある.

例えば図 7 の例では, `greenDao` といったタグ文字列や, 文字列定数中の `tables`, `schema`, `version` といった単語は他のログ関数呼出しでも利用されており, これらの単語のみをパターンとすると特定ができなくなる.

本研究の提案するソースコードの解析と単語の分割に基づいた推測は, ログ関数呼出し式の引数として合成される文字列定数に含まれる単語の一部がログ出力に現れる場合に, 単純なテキスト検索と比べて有効に働く. ソースコードの構造やログの文字列によっては

```
21 | public static final int SCHEMA_VERSION = 4;
47 | Log.i("greenDAO", "Creating tables for
   |             schema version " + SCHEMA_VERSION);
```

図 7　定数の参照がログ出力関数に含まれているコード

定数の定義箇所とログ関数呼出しの箇所が離れていたり, 複数の箇所で定数が使われている可能性があるため, 改めてログ関数呼出し式を探す必要がある.

また, 提案手法では, ログ出力行全体を自動的に単語に分割して推測を行うため, パターンとなる文字列を利用者が推測する必要がない.

4.1.4　提案したアルゴリズムの限界

4.1 節の計測対象において, 候補の推測が正確に行えなかった場合について分析する.

多くの場合, 推測が失敗した箇所は `android.support` や `com.google.android` のサブパッケージに含まれるクラスのログ関数呼出しで, これらは Android の標準 SDK に存在する追加的なライブラリ[*6]のコードである. これらの箇所は実際の開発ではアプリケーション開発者が参照することは稀であり, LogChamber ではパッケージ名で除外することができる. 例えば, Chapel Hill Transit アプリの 683 箇所のログ関数呼出しのうち, 267 箇所 (SR = 0.61) が失敗しているが, 標準 SDK の追加ライブラリ以外の箇所は 18 箇所 (約 2%) であった. 他のアプリケーションについても同様の傾向にある.

これらの失敗した原因は, 3.3 節で示したアルゴリズムによって得られる単語リストが同一となる複数の候補が存在するためである (単語リストが異なり, かつ一致度が同一になるような場合は, 我々の分析した範囲では見つからなかった.). 例えば, Tacere アプリには図 8 に示すような 2 つの異なる箇所の同一の内容のログ関数呼出しがある.

また, 単語リストが同一になる複数の候補が存在するケースとして, 全く定数の利用がなく単語が抽出できない (またはタグの 1 単語のみの) 場合がある. これは例えば, Tacera アプリの `AboutLicenseActivity` クラスに含まれる図 9 に示すような, ログメッセージの全体を動的な値で生成する場合である. これらの場合も, 結果として単語列が同じになる複数のログ関数呼出し箇所が生じるため, 提案手法では推測に失敗する.

また, 単語が抽出できない例として, Tacere アプリの `IabHelper` クラスに含まれる `logDebug` メソッド (図 10) のような, ログ関数を呼出すメソッドをアプリケーションで

[*6] これらは標準 SDK の基本的なクラスとは別に, アプリケーションごとに選択的に利用され, したがって APK ファイルに含まれる.

```
 70  public static PublicKey generatePublicKey
 78      Log.e(TAG, "Invalid key specification.");
109      Log.e(TAG, "Invalid key specification.");
```

図 8　同一のログ関数呼出しの例

```
45  } catch (FileNotFoundException e) {
46    Log.e(TAG, e.getMessage());
47  } catch (IOException e) {
48    Log.e(TAG, e.getMessage());
```

図 9　文字列定数をメッセージの本体として利用しないログ関数呼出し

```
1  void logDebug(String msg) {
2   if (mDebugLog) Log.d(mDebugTag, msg);
3  }
```

図 10　定数を参照しないログ出力関数定義

```
225  Log.d(Constants.TAG, "
       "resolved url is empty.  Url is:  " + url);
379  Log.e(Constants.TAG, responseCode +
       " url:" + urlAsString + " resolved:" + newUrl);
```

図 11　同じ単語があるログ出力関数定義

独自に定義する場合があった. このような場合は, 3.2 節で説明したように, logDebug メソッドをログ出力関数として抽出の対象にすることで正しく推測が可能となる[*7].

以上のような, 同一の単語リストとなるログ関数呼出し箇所が複数ある場合, 候補をログ関数呼出しを修正せずに, ログ出力から特定することは困難で, 将来的な課題である. ソースコード自体を修正することが可能であれば, ログ関数呼出しに特定可能な情報 (例えばログ出力にクラスファイルに埋め込まれた行番号) を付加する手法が考えられる. また, ユーザーインターフェース上で複数の候補があることを示す, といった対応も考えられる.

[*7] 関数 4.1 節の実験では網羅的な調査のため, アプリケーションごとの追加のログ出力関数の指定は行わなかった

表 3　候補数を変化させた場合の成功率

LPs	SR	Words
1	1.00	5
10	1.00	138
25	0.84	139
26	0.80	138
37	0.84	209
55	0.83	317

4.1.5　ログ関数呼出し箇所の相互の影響

ここでは, あるログ関数呼出しとそこから出力されるログ行に対して、推測候補の成功率 (SR) が他のログ関数呼出しの記述によって, どの程度影響を受けるかについて述べる.

ログ関数呼出しの数を変化させた場合に, 3.4 節で示した一致度がどのように変化するかを調べた. 対象として Lightning Browser アプリを選び, 候補となるログ関数呼出しとして抽出する式の数を適当に変化させ, 成功率 (SR) を計測した. 具体的には, 候補数 LPs を 1, 26, 37, 55(アプリ中のライブラリを除くすべての候補) と変化させた.

表 3 は結果である. 表の LPs は候補数を表し, Words は候補の全単語数, SR は成功率である.

結果から, 複数の候補がある場合は成功率が概ね 80% で, 3–5% 程度の範囲内で変化している. よって候補の変化が推測の結果に, 大きな影響を与えていないと考えられる.

図 11 は `HtmlFetcher.java` ファイルのコードである. 379 行目のログ出力関数が候補数 1, 26 個の際は, 正しく推測できていた. しかし 37, 55 個の候補の際には推測に失敗していることがわかった. これは候補に 225 行目が追加された時に発生しており, 原因としては`"url,resolved"`という単語リストに対して, 3.4 節のアルゴリズムの計算の結果同じスコアになってしまうため, 推測に失敗していたものである. これは, 4.1.4 節で示した限界と同様に同じ単語が含まれているケースであると同様である.

4.2　意図しない動作の修正作業による有効性の確認

LogChamber の有効性を確認するため, 実際のアプリケーションのデバッグを行った. 実験対象として Web ブラウザアプリケーション Lightning Browser を選んだ. このアプ

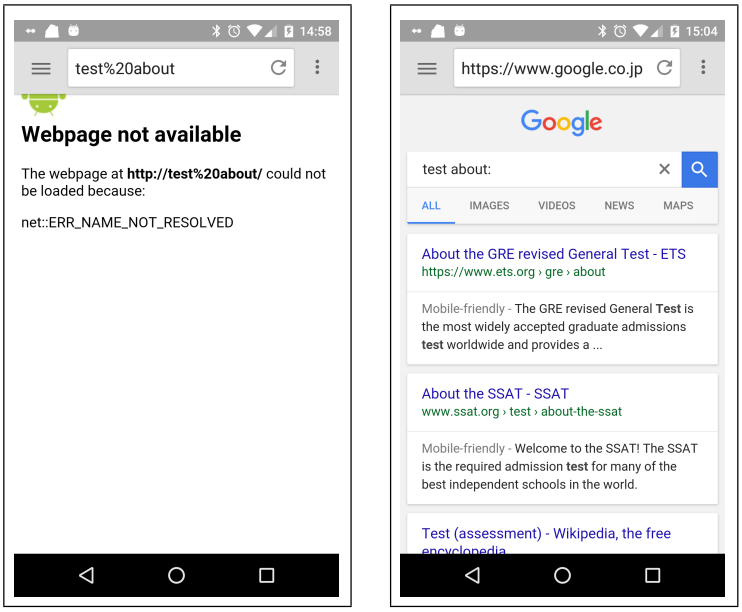

図 12　アプリケーションのエラー画面 (左) と正常な動作の画面 (右)

リケーションは実際に一般に配布が行なわれている[*8]典型的な Android アプリケーションで, ある程度の規模を持ち, かつソースコードが公開されている[*9]ため, 対象として適当と判断した.

有効性確認の実験には, 実際にアプリケーションの使用者から報告されたプログラマの意図しない動作として, Web ブラウザの検索が行えないという未解決の不具合 (Issue #127)[*10]を選択した. この意図しない動作は “cannot search words with a leading “about:”” というタイトルで報告され, アプリケーションの開発者は “bug” というラベルを設定しており, 著者らはソースコード上の未解決の意図しない動作であると推定した.

Lightning Browser は一般的な Web ブラウザと同様に画面上部にアドレスバーを表示し,URL を入力すると対応したページへ遷移する. また, アドレスバーに検索文字列を入力すると検索サイトによる検索結果のページへ遷移することができる.

対象とした意図しない動作は “`about:`” を含む文字列をアドレスバーに入力すると, 検索に失敗し空白のページを表示する動作をとる (図 12 左側). 期待される正常な動作は, 検索サイトにより文字列 “`test about:`” を検索した結果の表示である (図 12 右側). 今回

[*8] https://play.google.com/store/apps/details?id=acr.browser.barebones&hl=ja
[*9] https://github.com/anthonycr/Lightning-Browser
[*10] https://github.com/anthonycr/Lightning-Browser/issues/127

の意図しない動作は, ユーザーがアドレスバーに “test about:” と入力した際の意図しない動作を修正する.

実験に利用した対象アプリケーションのソースコードの版[*11]は約 9000 行の Java ソースコードからなり, ライブラリを除くアプリケーション内のログ呼出し箇所は 55 箇所であった.

著者らは LogChamber を適用し, 事前に対象のソースコードの内容に関する予備知識を持たずに, この意図しない動作のソースコード上の箇所を特定する作業を行った. 意図しない動作を発見した後に, 作業において LogChamber がどのように働いたかを分析した. さらに修正のためのコードをアプリケーションの開発者に実際に報告し, 修正が受け入れられるかを確認した.

4.2.1 ログ関数呼出しの追加

著者らは, まずアプリケーションにもともと挿入されている 55 箇所のログ関数呼出しから対象の意図しない動作の特定を試みた. アプリケーション起動時にはログ出力が 92 行, “test about:” という文字列を入力した際に取得できたログ出力は 26 行である. ただし, この中にはシステムのログが含まれており Lightning Browser が出力しているログ出力は 2 行であった。(図 13, I/Lightning ではじまるログ出力. なおここでは発生時刻やプロセス ID の情報は省略している.)

図 13 のログ出力の 5 行目の直前にアドレスバーに “test about:” を入力した. つまり, 5 行目以降が入力によって生成されたログ出力である. アプリケーションの出力した 4,5 行目の onResume と onPause は, LogChamber の推測結果からは BrowserActivity.java で生じることがわかり, ソースコードの閲覧から対応するログ関数呼出しを特定した. これらはアプリケーションの画面のライフサイクルの状態を表すログ出力であり, 意図しない動作のヒントを表すものではなく, これらのログ出力から意図しない動作の箇所特定は困難であると判断した.

これらのログ出力を行うログ関数呼出しは, 今回の意図しない動作に対して関連がある変数やログ関数呼出しがなかったため, 実行時ログから対象の意図しない動作を特定するために必要な情報が不足していた.

次に, 表 4 に示す 5 箇所のログ関数呼出しを新たに挿入し, 再度同じ入力に対して LogChamber を適用することで意図しない動作の特定を試みた. 著者らは, ソースコードのファイルを階層的に閲覧することで, 今回の意図しない動作の対象となりうる箇

[*11] commit ID: a2f2fbc

```
1  W/Resources Converting to string:  TypedValue
     {t=0x5/d=0x601 a=2 r=0x7f090011}
2  W/Resources Converting to string:  TypedValue
     {t=0x5/d=0xa01 a=2 r=0x7f09000e}
3  W/cr.BindingManager
     Cannot call determinedVisibility() -
     never saw a connection for the pid:  5488
4  D/Lightning onPause
5  I/Lightning onResume
6  E/ProxyChangeListener Using no proxy
     configuration due to
     exception:java.lang.NullPointerException:
     Attempt to invoke virtual method'java.lang.Object
     android.os.Bundle.get(java.lang.String)'
     on a null object reference
7  W/art?  Suspending all threads took:  5.370ms
8  W/art?  Suspending all threads took:  67.753ms
9  W/BindingManager Cannot call determinedVisibility()
     - never saw a connection for the pid:  6405
10 I/Lightning onPause
```

図 13　追加的なログ関数呼出し挿入前の Lightning Browser の実行時ログの例

所は, URL 問い合わせや検索に関連すると予測した. 簡単なファイルの閲覧により, `BrowserActivity.java` と `SearchAdapter.java` 中に URL の検索問い合わせに関する箇所を発見した. 従ってそれらの中で使用されている (表 4 で示した)`query` や `isSearch` といった値を追加的なログ出力とすることとした. その結果, 取得した実行時ログの一部が図 14 である.

4.2.2　意図しない動作の発見手順

図 14 の 5 行目は新たに追加したログ関数呼出しから生成された行であり,

```
isSearch = false
```

表 4　新たに追加したログ関数呼出しコード

ファイル名 (対象メソッド) 行数	挿入コード
BrowserActivity.java (searchTheWeb) 1558 行目	Log.d(Constants.TAG, "searchTheWeb query =" +query);
BrowserActivity.java (searchTheWeb) 1579 行目	Log.d(Constants.TAG, "isSearch = "+ isSearch);
BrowserActivity.java (searchTheWeb) 981 行目	Log.d(Constants.TAG, "findInPage query =" +query);
SearchAdapter.java (doInBackground) 296 行目	Log.d(Constants.TAG, "query ="+ query);
SearchAdapter.java (doInBackground) 358 行目	Log.d(Constants.TAG, "downloadSuggestionsForQuery query=" +query);

となっている. これは入力した文字列が検索のための文字列ではなく, URL として扱われることを示している. そのためアプリケーションは http://test%20about の URL にアクセスしようとする. そのため, この行は不正であり, 正しくは isSearch = true になると推定した.

LogChamber はこの行を出力した箇所は図 15 の BrowserActivity.java の 1579 行目と推測し, 図 16 のような表示を行った. 図 15 は当該箇所周辺の意図しない動作に関わるソースコードを示している. 1579 行目では変数 isSearch から生成した文字列を, ログ関数である Log.d メソッドの引数に与えている. isSearch は前方の 1573 行目に代入文があり, query と containsPeriod および aboutScheme の 3 つの変数から決まることがわかる.

これらの変数名から query が入力された文字列で, containsPeriod は入力がピリオド (.) を含むかを示し, そして aboutScheme は “about:” で始まる入力かどうかを示すと推定できる.

LogChamber は, 図 14, 4 行目のログ行

```
searchTheWeb query = test about:
```

```
1 I/Lightning onResume
2 E/ProxyChangeListener Using no proxy
    configuration due to
    exception:java.lang.NullPointerException:
    Attempt to invoke virtual method'java.lang.Object
    android.os.Bundle.get(java.lang.String)'
    on a null object reference
3 D/Lightning downloadSuggestionsForQuery query
     = test about:
4 D/Lightning searchTheWeb query = test about:
5 D/Lightning isSearch = false
6 W/BindingManager Cannot call determinedVisibility()
    - never saw a connection for the pid:  6405
7 I/Lightning onPause
```

図 14　追加的なログ関数呼出し挿入後の Lightning Browser の実行時ログの例

について, 追加した 1558 行のログ関数呼出しに対応することを推測し, query 変数の値が "test about:"であることを示している (図 17). この結果から, query が入力された文字列であることが明らかであり, query 自体の値は正常であることがわかる.

次に意図しない動作と関連が高いのは aboutScheme であることが名前から推定され, その値は前方の 1569 行目で決定されることがわかる.

4.2.3　意図しない動作の修正

実際に 1573 行目から aboutScheme の利用を削除し,

```
boolean isSearch = (query.contains(" ") || !containsPeriod);
```

のように変更して再実行したところ, アプリケーションは図 12 右側の表示のような期待する正常の動作を行った. 変数 aboutScheme が不正な値であることが分かった.

aboutScheme は図 15 の 1569 行目から,

```
query.contains("about:")
```

の呼出しによって生成されるので, この式を

```
1558 Log.d(Constants.TAG, "searchTheWeb query =" +query);
     ...
1567 boolean isIPAddress = (TextUtils.isDigitsOnly(query.replace(".", ""))
     && (query.replace(".", "").length() >= 4) &&     query.contains("."));
1569  boolean aboutScheme =  query.contains("about:");
1570  boolean validURL = (query.startsWith("ftp://") ||
      query.startsWith(Constants.HTTP)
     ...
1573  boolean isSearch =   ((query.contains(" ") || !containsPeriod)
      && !aboutScheme);
     ...
1579 Log.d(Constants.TAG,"isSearch = " + isSearch);
1580  if (isSearch)
```

図 15　意図しない動作箇所のコード

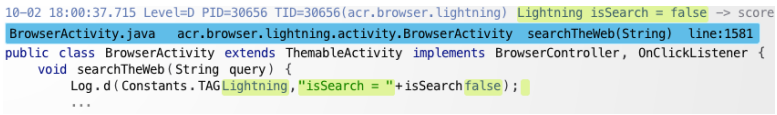

図 16　isSearch 変数に対する推測結果の表示

図 17　query 変数に対する推測結果の表示

```
query.startsWith("about:")
```

のように変更した修正パッチを作成し, 開発者に報告したところ, バグ修正として受け入れられた[*12].

[*12] commit ID: 33eb739

4.2.4 LogChamber の有効性

この実験では, LogChamber が実際の意図しない動作修正作業に貢献する事例を示した.

当初はログ出力から得られる情報が不足していたが, 一方で LogChamber により既存のログ出力の箇所を特定し, それらが意図しない動作と無関係であることを理解するために役立っている (4.2.1 節).

また, 新たなログ関数呼出しを追加したログ出力から, その出力箇所を特定し, LogChamber が表示した変数の値を確認することで, 意図しない動作の特定を行った (4.2.2 節).

LogChamber がなければ, これらの作業をすべて開発者が行う必要があり, 修正作業により時間がかかるものと予想される.

第 5 章　関連研究

ここでは LogChamber に関連する先行研究について議論する.

Yuan らの提案した SherLog [14] はプログラムの静的解析を用いて実行時ログから実行時の動作を推測するツールであり, LogChamber と同様のアイデアに基づいている. SherLog は C のプログラムが出力する実行時ログとそのプログラムのソースコードを解析し, 実行時の関数呼出しの流れの推測 (Path Inference) と変数の値の推測 (Value Inference) を行う.

一方, 現在の LogChamber はこれらの推測は行わない. これは 1) Android アプリケーションはイベント駆動の GUI アプリケーションであり, イベントハンドラのメソッドによる独立した実行が多いので, メソッド呼出しの流れや値の流れの推測がうまく働かない, 2) 大域的な解析に時間がかかるため, 即時の GUI 上での表示ができない, という理由のためである. SherLog はとくに単一のログ出力と対応するログ関数呼出しの推測ではなく, 関数呼出しの流れを含めたより大域的な推測を重視しており, そのままではモバイルアプリケーションのような, イベントループ中で繰り返し実行されるコールバックのイベントハンドラに対してはうまく働かないと予想される. 本研究の提案する LogChamber は単一のログ出力を個別に推測し, IDE 上の表示としてログ出力が行われると同時に推測候補を視覚的に表示することに成功している.

同様に, ログの解析に関する研究として Mariani らの手法 [9], Wei らの手法 [12, 13], LogEnhancement [15] AutoLog [16], lprof [17] などがあるが, 本研究は主にモバイルアプリケーションでの高速な (即時の) ログの推測に重点を置いている.

実行時の負荷をさける手法として, 記号実行 (Symbolic Execution) [6] が提案され, 意図しない動作の発見や検証に応用されている [2, 4]. 記号実行によりモバイルアプリケーションのデバッグを行う研究 [5, 10] があるが, 端末によって動作が異なるモバイルアプリケーションでは, それらの端末の違いを静的な実行時に再現することが難しく, 本研究が対象とする特定の端末上で再現する意図しない動作には対応しにくい .

モバイルアプリケーションの解析に関する研究として, FlowDroid [1], Scandroid [3], Klieber らの手法 [7] は, 正確な制御の流れの静的解析手法を提案し, 脆弱性の発見などに応用ができる. FlowDroid は Java アプリケーションの解析フレームワークである Soot [11] を拡張し, Android アプリケーションの解析に特化し, 2 つのメソッド間で特定の変数の影響がありうるかを調べることができる. 本研究では LogChamber の実装に

FlowDroid による拡張の APK ファイル読み込み機能を一部利用しており, 将来的にコールグラフの検索などで応用が可能と考えている.

第 6 章　まとめ

本研究では, モバイルアプリケーションのデバッグの問題を解決するため, ログ出力の解析を行い出力箇所を特定するツール LogChamber を提案した. LogChamber はモバイルアプリケーションに書かれがちな, 場当たり的で再現性の低いログ関数呼出しに対して, それらを手動で管理することなく, 自動的にログ出力から対応するログ関数呼出しを推測することができる. また, LogChamber は実際の IDE 上にプラグインとして実装され, デバッグ実行中のアプリケーションに接続し, ログ出力をその場で推測し, 対応するソースコードにログ関数呼出しに渡された式に対して該当するログ行の文字列を表示することができる. これらの結果, モバイルアプリケーションのデバッグの時間短縮に貢献できる.

実験によって実際の複数のモバイルアプリケーションに対して LogChamber を適用し, ある程度大きな規模のプログラムでも有効に働き, 実用的な速度で動作することを示した. また, 実際のアプリケーションの未解決の意図しない動作を特定し修正する実験を行い, LogChamber が有効に働く事例を示した.

本稿で示した手法は, 多くの場合実用的にログ関数呼出しの箇所を推測することができるが, 場合によっては誤った箇所を推測する可能性もある. 開発者がログ関数呼出しに利用する単語に注意を払うことで改善ができるが, 将来的にはそのようなログ関数呼出しのユニークさをあらかじめ評価したり, 単語の候補を提示する機能の開発が考えられる.

また, 現在の手法は複数のログ行間の制御の流れの解析などは行っていないが, 将来的な拡張として効率的に大域的なデータや制御の流れを特定し, プログラムの実行履歴を復元するような手法を開発したいと考えている.

謝辞

本論文は、筆者が金沢大学自然科学研究科電子情報科学専攻博士課程に在学中に行った研究を纏めたものである。本研究を進めるにあたり, 素晴らしい研究環境を与えて頂いたこととともに, 多大なる御指導・御教示を賜りました櫻井 孝平 助教に心から誠意を表わすとともに厚く御礼申し上げます. 本論文を査読して頂くとともに、研究を進めていくにあたり終始懇切丁寧なご指導を頂きました山根 智 教授に深く感謝いたします。日々, 研究室での学生生活において、お世話になりました山根研究室の皆様に感謝の意を表します. 最後に, 今日に至るまでの学生生活を様々な面から支えていただいた, 父母や弟に心から感謝の意を表します.

参考文献

[1] 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, *SIGPLAN Not.*, Vol. 49, No. 6, pp. 259–269 (2014).

[2] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P. and Sheth, A. N.: TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones, *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, Berkeley, CA, USA, USENIX Association, pp. 1–6 (2010).

[3] Fuchs, A. P., Chaudhuri, A. and Foster, J. S.: Scandroid: Automated security certification of android applications, Technical report, University of Maryland (2009).

[4] Jensen, C. S., Prasad, M. R. and Møller, A.: Automated Testing with Targeted Event Sequence Generation, *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, New York, NY, USA, ACM, pp. 67–77 (2013).

[5] Jeon, J., Micinski, K. K. and Foster, J. S.: SymDroid: Symbolic execution for Dalvik bytecode, Technical report, Department of Computer Science, University of Maryland, College Park (2012).

[6] King, J. C.: Symbolic Execution and Program Testing, *Commun. ACM*, Vol. 19, No. 7, pp. 385–394 (1976).

[7] Klieber, W., Flynn, L., Bhosale, A., Jia, L. and Bauer, L.: Android Taint Flow Analysis for App Sets, *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*

[8] Lewis, B.: Debugging Backwards in Time, *Proceedings of the Fifth International Workshop on Automated Debugging*, AADEBUG'03 (2003).

[9] Mariani, L. and Pastore, F.: Automated identification of failure causes in system logs, *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, IEEE, pp. 117–126 (2008).

[10] Mirzaei, N., Malek, S., Păsăreanu, C. S., Esfahani, N. and Mahmood, R.: Testing Android Apps Through Symbolic Execution, *SIGSOFT Softw. Eng. Notes*, Vol. 37, No. 6, pp. 1–5 (2012).

[11] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V.: Soot - a Java Bytecode Optimization Framework, *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, IBM Press, pp. 13– (1999).

[12] Xu, W., Huang, L., Fox, A., Patterson, D. and Jordan, M.: Experience Mining Google's Production Console Logs, *Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, SLAML'10, Berkeley, CA, USA, USENIX Association, pp. 5–5 (2010).

[13] Xu, W., Huang, L., Fox, A., Patterson, D. and Jordan, M. I.: Detecting Large-scale System Problems by Mining Console Logs, *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, New York, NY, USA, ACM, pp. 117–132 (2009).

[14] Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y. and Pasupathy, S.: SherLog: Error Diagnosis by Connecting Clues from Run-time Logs, *SIGARCH Comput. Archit. News*, Vol. 38, No. 1, pp. 143–154 (2010).

[15] Yuan, D., Zheng, J., Park, S., Zhou, Y. and Savage, S.: Improving Software Diagnosability via Log Enhancement, *SIGARCH Comput. Archit. News*, Vol. 39, No. 1, pp. 3–14 (2011).

[16] Zhang, C., Guo, Z., Wu, M., Lu, L., Fan, Y., Zhao, J. and Zhang, Z.: AutoLog: Facing Log Redundancy and Insufficiency, *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, New York, NY, USA, ACM, pp. 10:1–10:5 (2011).

[17] Zhao, X., Zhang, Y., Lion, D., Ullah, M. F., Luo, Y., Yuan, D. and Stumm, M.: Lprof: A Non-intrusive Request Flow Profiler for Distributed Systems, *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Berkeley, CA, USA, USENIX Association, pp. 629–644 (2014).