修士学位論文

題目

実行履歴の区間の照合に基づいた
類似クラスおよび類似メソッドの検出手法

指導教員
井上 克郎 教授

報告者
井岡 正和

平成 25 年 2 月 5 日

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 ソフトウェア工学講座

平成 24 年度 修士学位論文

実行履歴の区間の照合に基づいた類似クラスおよび類似メソッドの検出手法

井岡 正和

内容梗概

近年，ソースコードの剽窃が増加している．ソースコードの剽窃とは，ある者が著作者の意図に反してソースコードを再利用することをいう．また，ソースコードにおける剽窃は，ソースコード全体が剽窃される場合と，クラスやメソッド等のソースコードの一部が剽窃される場合がある．ソースコードの一部が剽窃された場合に，剽窃された部分の特定に使用できる技術として，ソースコード間に存在する重複部分を特定するコードクローン検出手法が挙げられる．一方で，ソースコードを書き換え，内容の理解を困難にする難読化技術が存在する．ソースコードに難読化技術を施すと，メソッドが別のクラスに移動される等，ソースコードの静的な構造が改変される．そのため，剽窃を行った者がソースコードに難読化技術を適用すると，既存のコードクローン検出手法では剽窃の特定が困難になる．

そこで，本研究では，難読化の影響が少ないプログラムの実行履歴を分析することで，類似クラスや類似メソッドを検出する手法を提案する．提案手法では，プログラムの実行履歴を複数のフェイズに分割し，各フェイズのメソッド呼び出し列を比較することで，類似クラス，類似メソッドを検出する．

提案手法の難読化に対する耐性を確認するために，実際の Java で開発されたソフトウェアのソースコードを難読化し，提案手法を適用した結果，難読化前後で同一のソフトウェアを識別できることを確認できた．また，剽窃が疑われる Android アプリケーションにも提案手法を適用し，Android アプリケーションがソースコードレベルの剽窃であるかを識別できることを確認できた．

主な用語

動的解析 (Dynamic Analysis)
難読化 (Obfuscation)
剽窃 (Plagiarism)

目 次

1 はじめに **3**

2 背景 **5**

2.1 コードクローンとその検出 5
2.2 プログラムの実行履歴分析 6
2.3 渡邊らのフェイズ分割手法 6
2.4 ソースコードの難読化技術 9
2.5 既存の剽窃検出手法とその問題点 11

3 提案手法 **13**

3.1 プログラムの実行履歴のフェイズ分割 13
3.2 メソッド呼び出しの正規化 13
3.3 フェイズマッチング 15
3.4 クラス・メソッドマッチング 19

4 適用実験 **26**

4.1 難読化に対する耐性の検証 26
4.1.1 クラス・メソッド単位の類似性判定 27
4.1.2 コードクローン検出を用いた類似性判定 27
4.2 剽窃が疑われるソフトウェアへの適用 28
4.2.1 準備 28
4.2.2 関数電卓アプリケーション 28
4.2.3 履歴削除アプリケーション 30

5 考察 **33**

5.1 実験結果から得られた知見 33
5.2 妥当性への脅威 33

6 まとめと今後の課題 **36**

謝辞 **37**

参考文献 **38**

1　はじめに

近年，著作者の意図に反したコピーによる再利用等のソースコードの剽窃が増加している [17, 21]．ソースコードの剽窃は，ソースコード全体が剽窃される場合と，クラスやメソッド等のソースコードの一部が剽窃される場合がある．ソースコードの一部が剽窃された場合は，ソースコード間に存在する重複部分を特定するコードクローン検出手法を利用して，剽窃を検出することができる [5]．

一方で，ソースコードを書き換え，内容の理解を困難にする難読化技術が提案されている [10, 19]．剽窃を行った者がソースコードに難読化技術を適用すると，メソッドが別のクラスに移動される等，プログラムの静的な構造が改変される．そのため，トークン列や構文木の等価性に基づくコードクローン検出手法 [9] では，剽窃を特定することが難しい．また，システム依存グラフ [6] の等価性に基づいてコードクローンを検出することで，難読化に対応可能であると考えられるが，計算コストが非常に大きい．

ソフトウェアの特徴の中で難読化の影響が少ないものとして，プログラムの実行履歴が考えられる．プログラムの実行履歴は，実行時のオブジェクトの振る舞いを記録したもので，実行されたイベントが時系列順に並んでいる．API 呼び出しの系列や頻度といったプログラムの実行履歴上の特徴を用いて，ソフトウェア単位の剽窃を検出する手法 [16] が提案されている．しかし，プログラムの実行履歴からクラスやメソッドの特徴を抽出し，それらの剽窃の特定を行った研究は確認されていない．クラス単位やメソッド単位でも剽窃されるため，これらの剽窃を検出する手法が必要とされている．

そこで本研究では，プログラムの実行履歴から類似クラス，類似メソッドを検出する手法を提案する．提案手法は，2 つのプログラムの実行履歴を与えると，プログラムの実行履歴を機能的なまとまりであるフェイズに分割し，フェイズ間の類似度を計算して類似度の高いものから順にフェイズを対応付ける．そして，対応付けられたフェイズからクラス間，メソッド間の類似度を計算し，類似度の高いものを類似クラス，類似メソッドとして出力する．実行履歴の分析を行う提案手法は，静的な構造を変化させる難読化だけでなく，Java 言語や C#言語等のリフレクションを生成する難読化にも対応できる [19]．また，剽窃が疑われるソフトウェアのソースコードを入手できることはまれであるため，提案手法のように，実行ファイルのみで解析できることは重要であると考えられる．

提案手法の有効性を確認するために，提案手法をツールとして実装し 2 種類の適用実験を行った．まず，難読化に対する耐性を確認するために，実際の Java で開発されたソフトウェアのソースコードを難読化し提案手法を適用した結果，難読化前後で同一のソフトウェアを識別できることを確認した．さらに，剽窃が疑われる 2 つの Android アプリケーションに提案手法を適用し，これらの Android アプリケーションがソースコードレベルの剽窃であ

るかを識別できることを確認した．

以降，2 章では本研究に関連する用語の説明および，既存の剽窃検出手法とその問題点について述べる．そして，3 章では提案手法であるプログラムの実行履歴から類似クラス，類似メソッドを検出する手法について述べ，4 章では実際のソフトウェアに対して適用実験を行った結果を述べる．そして 5 章で実験に対する考察を行い，最後に 6 章でまとめと今後の課題について述べる．

2　背景

本研究の背景として，静的な剽窃検出に利用されているコードクローン検出手法及び，提案手法で用いるプログラムの実行履歴とフェイズ分割手法について述べる．また，剽窃を隠蔽することに用いられるソースコードの難読化技術についても述べる．既存の剽窃検出手法としては，静的な情報を用いた手法，動的な情報を用いた手法及び，それらの問題点について述べる．

2.1　コードクローンとその検出

コードクローンとは，ソースコード内の同一または類似するコード片のことをいい，主に既存コードのコピーアンドペーストや，定型処理の記述によって生成される [1, 11]．また，ソースコードが剽窃された際にも生成される．

コードクローンを検出する手法が，既存研究で数多く提案されている [7, 9, 12, 14, 18, 23]．コードクローン検出手法は，ソースコードの剽窃検出に利用されており，コードクローンの検出単位によって大きく以下の 5 つに分類できる．

- 行単位の検出
- 字句単位の検出
- 抽象構文木を用いた検出
- プログラム依存グラフを用いた検出
- メトリクスやフィンガープリント等，その他の技術を用いた検出

この分類において，上に記述したものほど高速にコードクローンを検出でき，下に記述したものほど差分を含んだコードクローン等の多様なコードクローンを検出することができる．

ここでは，コードクローン検出手法の例として，字句単位の検出にあたる CCFinder[9] について述べる．CCFinder は，プログラムのソースコード中に存在する極大クローンを検出し，その位置を出力する．この手順は，以下の 4 つのステップからなる．

Step 1. 字句解析 ソースコードを字句解析し，トークン列に変換する．

Step 2. 変換処理 実用上意味を持たないコードクローンを取り除くことや，些細な表現上の違いを吸収するためにトークン列を変換する．例えば，変数名は同一のトークンに置換されるので，変数名が違うコード片もコードクローンとして判定することができる．

Step 3. 検出処理 トークン列の中から，指定されたトークン列の長さ以上一致している部分をコードクローンとしてすべて検出する．

Step 4. 出力整形処理 検出したコードクローンのソースコード上の位置情報を出力する．

2.2 プログラムの実行履歴分析

プログラムの実行履歴とは，実行時のオブジェクトの振る舞いを記録したもので，実行されたイベントが時系列順に並んでいる．イベントは，実行されたオブジェクト間のメッセージ通信イベントであり，実行時刻やメッセージを送信したオブジェクトと受信したオブジェクト，メッセージの内容等の情報を保持している [22]．

プログラムの実行履歴を抽出するツールの 1 つとして Amida[20] がある．Amida のプロファイラ機能は，Java Virtual Machine Tool Interface(JVMTI) [1] で実装されたダイナミックリンクライブラリであり，Javassist[2] 等を介して Java のバイトコードを変換し，プログラムの実行時のイベントを実行履歴として記録するものである．イベントとしては，メソッドの呼び出し，復帰やフィールドの参照，定義等を検出することができる．

2.3 渡邊らのフェイズ分割手法

渡邊らは，1 つのプログラムの実行履歴を複数のフェイズに分割する手法を提案している [22]．フェイズとは，プログラムの実行履歴上から切り出された連続するイベント列のうち，「入出力処理」や「データベースアクセス」等の開発者にとって意味のある処理に対応するものを指す．

オブジェクト指向プログラムは，1 つの機能を実行する際に多数の中間データ用のオブジェクトを生成し，その機能の実行が終了した時点でそれらのオブジェクトの大半を破棄するという性質を持っている [13]．そのため，メソッド呼び出しイベントに関わったオブジェクトを Least Recently Used (LRU) キャッシュに登録していくことで，キャッシュの更新頻度からあるフェイズが終了し次のフェイズが開始したことを検知することができる．

フェイズ分割アルゴリズムを Algorithm 1 に示す．この手法は，入力としてプログラムの実行履歴から抽出されたメソッド呼び出しイベント列 $[e_1, e_2, ..., e_{last}]$ と，4 つのパラメータとしてキャッシュサイズ c，ウィンドウサイズ w，スコープサイズ m，閾値 $threshold$ を受け取る．メソッド呼び出しイベント e_k は，イベント時刻 k，呼び出し元オブジェクト ID $e_k.caller$，呼び出し先オブジェクト ID $e_k.callee$，コールスタックの深さ $e_k.callstack$ の 4 つの情報を持つ．

---
[1] JVMTI: `http://docs.oracle.com/javase/jp/6/technotes/guides/jvmti/`
[2] Javassist: `http://www.csg.is.titech.ac.jp/~chiba/javassist/`

Algorithm 1 フェイズ分割アルゴリズム [22]

procedure DetectPhases (

in [e_1, e_2, ..., e_{last}]: list of method call event;

in c, w, m: integer;

in $threshold$: double;

out P: set of integer);

1. LRUCache $C \leftarrow$ new $LRU_cache(c)$
2. integer[] $update \leftarrow []$
3. double[] $frequency \leftarrow []$
4. set of integer $P \leftarrow \phi$
5. **for** integer t **in** [1 ... $last$] **do**
6. boolean $b1 \leftarrow C.update(e_t.caller, t)$
7. boolean $b2 \leftarrow C.update(e_t.callee, t)$
8. **if** $b1 \vee b2$ **then** $update[t] \leftarrow 1$ **else** $update[t] \leftarrow 0$
9. double $flags \leftarrow 0$
10. **for** integer k **in** [max($t - w + 1$, 1) ... t] **do**
11. $flags \leftarrow flags + update[k]$
12. **end for**
13. $frequency[t] \leftarrow flags/w$
14. **if** $frequency[t] \geq threshold$ **then**
15. integer $min \leftarrow$ max($t - m + 1$, 1)
16. **for** integer k **in** [max($t - m + 1$, 1) ... t] **do**
17. **if** $e_{min}.callstack \geq e_x.callstack$ **then** $min \leftarrow k$
18. **end for**
19. $P \leftarrow P.add(min)$
20. **end if**
21. **end for**

Algorithm 1 の入力の 4 つのパラメータは以下の通りである．

キャッシュサイズ c Algorithm 1 の 1 行目で生成される固定長の LRU キャッシュのサイズを表す整数値である．最小値は 1 であり，最大値はプログラムの実行履歴に現れるオブジェクト数に等しい．キャッシュサイズは，1 つのフェイズを構成するとみなすオブジェクト群の要素数を表すため，検出されるフェイズの粒度に影響を与える．

ウィンドウサイズ w 各イベント時刻における LRU キャッシュの更新頻度 $frequency$ を計算する範囲を表すイベント数である (Algorithm 1 の 10，13 行)．最小値は 1 であり，最大値はプログラムの実行履歴に含まれるメソッド呼び出しイベントの数 (Algorithm 1 における $last$) に等しい．小さいほど検出フェイズ数が増加し，大きいほど検出フェイズ数が減少する．

スコープサイズ m フェイズ移行検知時に，新しいフェイズの開始点の探索範囲を指定するイベント数である (Algorithm 1 の 15，16 行)．最小値は 1 であり，最大値はプログラムの実行履歴に含まれるメソッド呼び出しイベント数に等しい．フェイズ移行が検知されるイベント時刻と対応するフェイズの開始点の距離は，他の 3 つのパラメータによって変化するため，スコープサイズを適切に設定する必要がある．

閾値 $threshold$ 各イベント時刻における LRU キャッシュの更新頻度 $frequency$ がフェイズ移行を表す下限値として用いる閾値である (Algorithm 1 の 14 行)．0 から 1 の実数値をとる．閾値が小さいほど検出フェイズ数が増加し，大きいほど検出フェイズ数が減少する．なお，ウィンドウサイズと違い，LRU キャッシュの更新頻度そのものには影響を与えない．

各フェイズの開始点を検出する手順は以下の 3 つのステップからなる．

Step 1. 動作するオブジェクト群の観測 (Algorithm 1 の 6~8 行) プログラムの実行履歴中のメソッド呼び出しイベントの呼び出し元オブジェクトと呼び出し先オブジェクトを，時系列順に LRU キャッシュC に追加し，各時刻におけるキャッシュの更新の有無を判定する．

Step 2. フェイズ移行の検知 (Algorithm 1 の 9~13 行) フェイズの移行を表す値として，履歴上の各イベント時刻 t における LRU キャッシュC の更新頻度 ($frequency(t)$) を次の式で定義する．

$$frequency(t) = \frac{\sum_{k=max(1,t-w+1)}^{t} update[k]}{w}$$

$update[k]$ は，イベント時刻 k でキャッシュが更新された場合に 1，更新されなかった場合に 0 の値をとる．$frequency(t)$ は，イベント時刻 t から過去 w 回のメソッド呼び出しイベント中でキャッシュの更新が生じたイベントの割合を表し，この値が閾値 $threshold$ より高い場合にフェイズ移行が生じたとみなす.

Step 3. フェイズ開始点の識別 (Algorithm 1 の 14~20 行) コールスタックの深さ情報を用いて，新しいフェイズの開始点となるイベントを識別する．新しいフェイズの開始点となるイベントは，コールスタックの深さが局所的最小値をとるものであるため，フェイズの移行が検知されたイベントから過去 m 回分のイベント列でコールスタックの深さを比較し，最小値をとるイベントをフェイズの開始点として検出する．なお，最小値をとるイベントが複数存在する場合は，最も新しいイベントを選択する．

すべてのメソッド呼び出しイベントに上記の処理を適用し，検出したフェイズの開始点のイベント時刻の列 $P = [t_1, t_2, ..., t_p]$ を出力する．なお，このアルゴリズムは，入力するプログラムの実行履歴のサイズにほぼ比例した時間コストで計算可能である．

適用実験では，企業で開発された 2 つの Java ソフトウェアに対して手法を適用し，この手法によって自動的に検出されるフェイズと，開発者が実行履歴を手動で読み解いて決定したフェイズの比較を行なっている．その結果，8 割程度のフェイズが正しく対応していることを確認している．

2.4 ソースコードの難読化技術

難読化とは，あるソースコードを実行結果が等価であり理解が困難なソースコードに変換することである [19]．難読化を施すことによって，ソースコードを不正な解析から保護することができる．一方で，不正にコピーしたソースコードに難読化を施すことによって，剽窃が隠蔽される恐れがある．難読化には様々な技術が存在するが，ここでは，名前変換，メソッド分散，インライン展開について述べる [2, 19].

名前変換 対象クラスに含まれるシンボル名 (クラス名，フィールド名，メソッド名等) の定義を変更し，意味のない名前にする技術である．定義を変更するため，システムが提供する API に含まれる名前を変更することはできない．例を図 1 (a) に示す．例ではシンボル名を a，b 等に変換している．

メソッド分散 対象クラスの集合からランダムに選択したメソッドを別のクラスに移動する技術である．例を図 1 (b) に示す．呼び出されるメソッドの修飾子を public static にし，呼び出す側はこのメソッドを静的に呼び出すように変更している．また，移動し

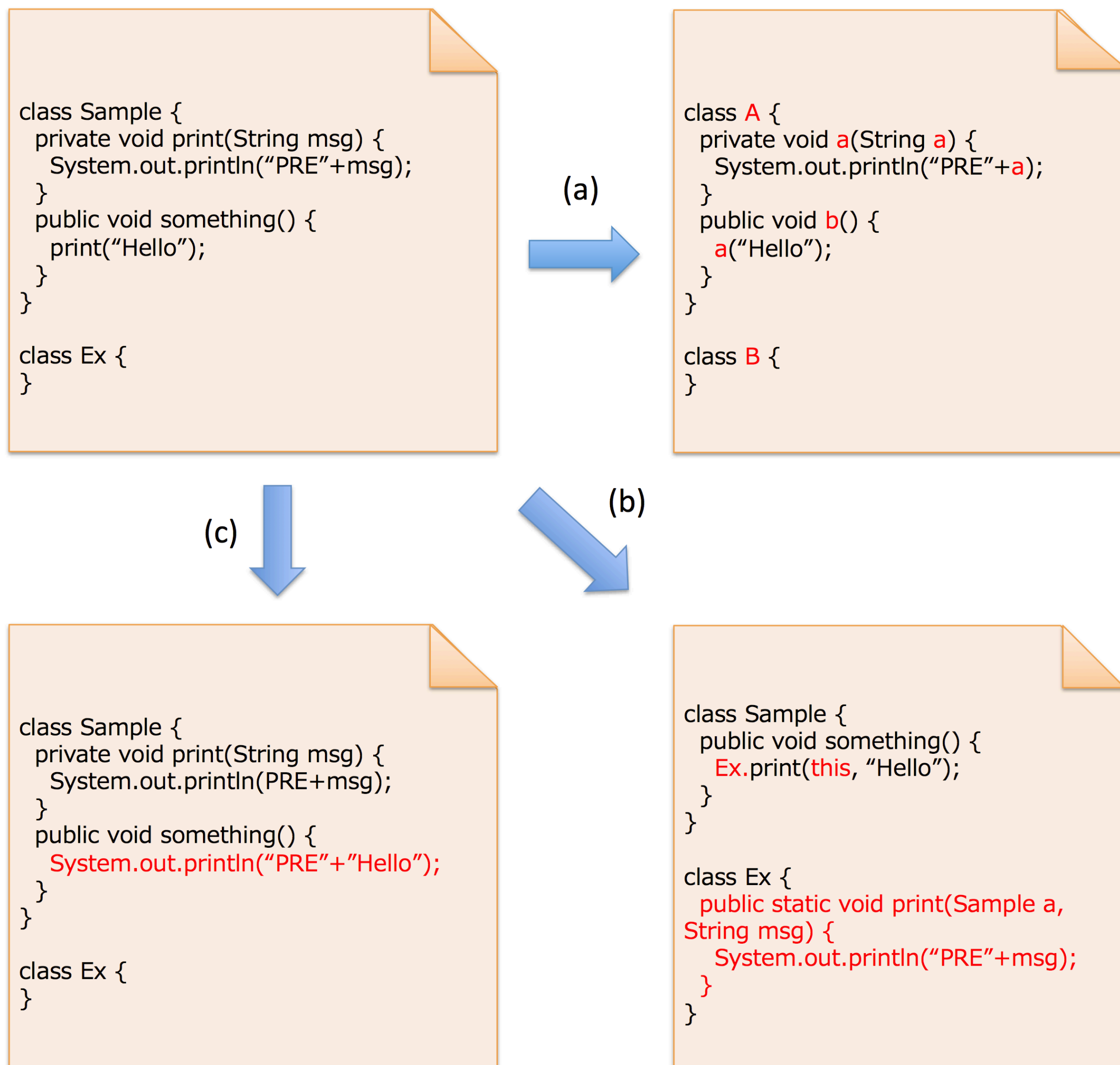


図 1: 難読化の例

たメソッドから移動元クラスのフィールドを参照できるように，引数に移動元のオブジェクトを渡すように変更している．

インライン展開 対象クラスの集合からランダムに選択したメソッド呼び出しにそのメソッドのコードを展開する技術である．例を図 1 (c) に示す．例では，something メソッド内の print メソッドの呼び出しに，print メソッドのコードを展開している．

多くの難読化技術が公開されており，誰でも容易に利用することができる[3] [4] [5]．さらに，ソースコードの偽装を行う手法が提案されている [10]．この手法では，プログラムの静的解析を困難にするために，見せかけのソースコードを作成し，実行時に元来のソースコードの内容が実行されるように自己書き換えを行う．

このような難読化技術を剽窃を行った者がソースコードに適用すると，既存のコードクローン検出技術では等価な処理をコードクローンとして識別することが困難である．

2.5 既存の剽窃検出手法とその問題点

Lim らは，制御フローグラフを用いた静的な剽窃検出手法を提案している [14]．この手法では，ソフトウェアの制御フローグラフから実際に実行されうるフローを求め，そのフローのバイトコードを比較することによって剽窃を検出する．ソフトウェアの静的な情報を用いているが，ソフトウェアに難読化が施された場合でも制御フローグラフは変化しにくいため，この手法は難読化にある程度対応している．評価実験では，他の剽窃検出手法との比較実験を行い，Lim らの手法が効果的で信頼できることを確認している．

Maskeri らは，リポジトリマイニング技術でコーディングスタイルを取得し，その情報を用いて剽窃の疑いのあるコードを検出する手法を提案している [15]．コーディングスタイルとしては，小文字，大文字の使い分け，空白，インデント，改行の仕方や，コメントの書き方等が挙げられる．この手法では，ある開発者のコーディングスタイルをリポジトリから取得して，その特徴を保存する．そして，リポジトリからその開発者が開発したとされるコードを取得し，その特徴とコーディングスタイルが異なる場合に，剽窃の疑いがあるコードと判定する．なお，あくまで剽窃の疑いがあるコードを検出する手法であり，最終的に剽窃と判断する場合には既存のコードクローン検出手法を用いる必要がある．適用実験として，Eclipse JDT-Core プロジェクトから他人のコードの再利用を検出できるか実験しており，コーディングスタイルの異なる箇所が他人のコードの再利用であることを特定している．

岡本らは，改ざんされにくい情報である Application Program Interface(API) の呼び出しに着目し，ソフトウェア単位の剽窃を検出する手法を提案している [16]．OS が API によって提供する機能としては，ファイル入出力機能，ユーザインターフェイス機能，同期機能等がある．この手法では，まず，あるソフトウェアをある入力を与えて実行し，そのソフトウェアの実行時の API の呼び出し履歴から，API の呼び出し順序と API の呼び出し頻度を抽出する．そして，抽出した 2 つの情報を比較して剽窃を検出する．なお，この手法は，Windows 環境で動作するプログラムとして実装されており，API の呼び出しをある程度利

[3]ProGuard: http://proguard.sourceforge.net/
[4]DashO: http://www.agtech.co.jp/products/preemptive/dasho/
[5]Allatori: http://www.allatori.com/

用している中規模以上のソフトウェアを対象としている．評価実験では，同じ用途の実用ソフトウェア郡を対象に手法を評価しており，あるソフトウェアとそのソフトウェアに等価な変換を施したものが同じであると判定できることと, 独立に実装されたソフトウェアを区別できることを確認している．さらに別の実験で，コンパイラの最適化に対する手法の耐性についても評価しており，異なるコンパイラやコンパイルオプションによる手法への影響は軽微であることを確認している．

Lim らの手法等の既存の静的な手法では，静的な構造を変化させる難読化だけでなく，Java 言語や C#言語等のリフレクションを生成する難読化に対応することは難しい．一方で岡本らの手法等の既存の動的な手法に関しては，プログラムの実行履歴全体同士を比較する手法しか存在しない．そのため，小さい単位のマッチングが困難であり，クラス単位やメソッド単位の剽窃は対象としていない．しかし，クラス単位やメソッド単位でも剽窃され，難読化を施されることがあるため，これらの剽窃を検出する必要がある．

3 提案手法

本研究では，2 つのプログラムの実行履歴を与えると，その 2 つのプログラムの実行履歴から類似クラス，類似メソッドを検出する手法を提案する．この手法では，プログラムの実行履歴を対象としているため，ソースコードが入手できない場合や，プログラムを難読化された場合でも剽窃を検出することができる．なお，提案手法ではプログラムの実行履歴中のメソッド呼び出し列を比較するので，ここではプログラムの実行履歴として,「メソッドの呼び出し」と「コンストラクタの呼び出し」のみをイベントとして用いる．以降，これらのイベントをまとめて「メソッド呼び出し」と呼ぶ.

図 2 に提案手法の概要を示す．提案手法は，以下のステップからなる．

Step 1. フェイズ分割 与えられたプログラムの実行履歴をフェイズに分割する．

Step 2. 正規化 各フェイズのメソッド呼び出しの正規化を行う．

Step 3. フェイズマッチング 動的計画法を用いて各フェイズを比較し対応付ける．

Step 4. クラス・メソッドマッチング フェイズ間の対応付けを用いてクラスおよびメソッドを対応付ける．

以降，各ステップについて詳述する．

3.1 プログラムの実行履歴のフェイズ分割

プログラムの実行履歴を機能単位で比較することによって，クラス単位やメソッド単位の比較が行いやすくなる．また，比較の計算コストを小さくすることができる．以上の理由で，与えられたプログラムの実行履歴を，2.3 節で挙げたフェイズ分割手法を用いて複数のフェイズに分割する．なお，スレッドごとにまとまった処理が実行されるため，フェイズ分割をスレッドごとに行う．

3.2 メソッド呼び出しの正規化

同型の類似フェイズや難読化に対応するために，各フェイズについて 2 種類の正規化を行う．

メソッド呼び出し列の正規化 メソッド呼び出しの繰り返し回数のみが異なる同型のフェイズを類似したフェイズとして検出するために，2 回以上の連続した同じメソッド呼び出しは 2 回のメソッド呼び出しとする．

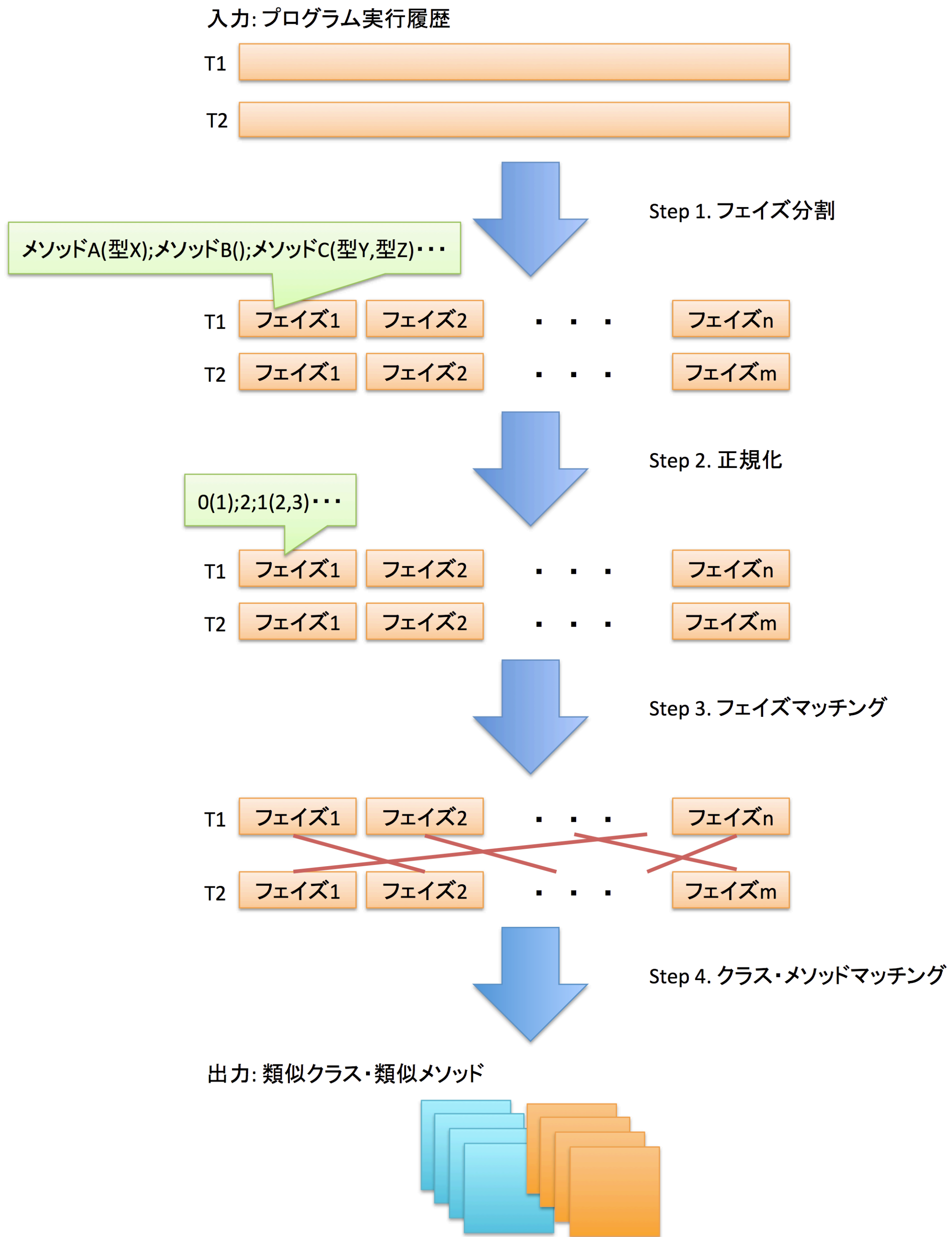


図 2: 提案手法の概要

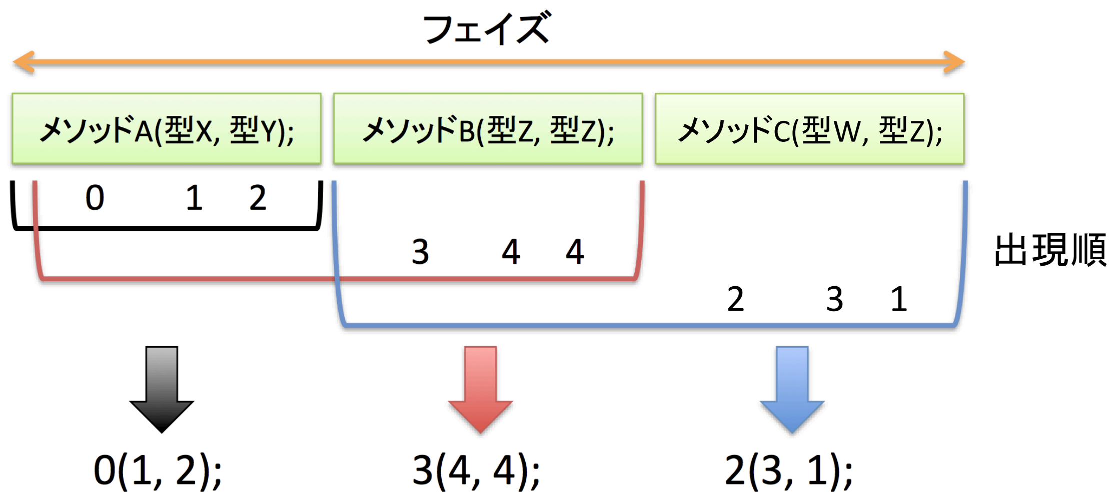


図 3: メソッド呼び出し文の正規化の例

メソッド呼び出し文の正規化 難読化によってメソッドのシグネチャが意味を持たなくなるため，メソッド呼び出しの系列を，各シグネチャのメソッド呼び出し内の出現順の系列に変換する．その際，一つ前のメソッド呼び出しを出現の起点とすることにより，メソッド呼び出し間の関連を考慮する (ただし，フェイズ先頭のメソッド呼び出し文は除く)．例を図 3 に示す．メソッド呼び出し A(X, Y) については，一つ前にメソッド呼び出しがないので，メソッド呼び出し A(X, Y) のみのシグネチャA，X，Y の出現順 0，1，2 を利用した「0(1, 2);」に変換する．メソッド呼び出し B(Z, Z) については，一つ前のメソッド呼び出しである A(X, Y) を起点として，B，Z の出現順は 3 番目，4 番目となるので「3(4, 4);」となる．メソッド呼び出し C(W, Z) についても同様で「2(3, 1);」となる．

3.3 フェイズマッチング

フェイズの長さ (メソッド呼び出しの数) が短いものは異なる処理のフェイズであってもメソッド呼び出し列が重複しやすいので，フェイズの長さが閾値未満のものを検出対象から除外する．そして，動的計画法を用いた類似文字列マッチングアルゴリズム [24] を使用してフェイズの比較を行う．このアルゴリズムは，入力として 2 つの文字列を与え，一方の文字列に，1 文字削除，1 文字挿入，1 文字置換という 3 つの操作を最低何回行ってもう一方と同一の文字列へと変化させられるかを調べることで，2 つの文字列の類似度を求めることができる．

提案手法では，出現順に変換した 1 メソッド呼び出しを 1 文字に符号化してこのアルゴ

リズムを適用する．このアルゴリズムを用いることで，フェイズ間の類似度と，フェイズ間のメソッド呼び出しの対応関係を得ることができる．フェイズ α 中に出現する全メソッド呼び出しの数を $N_{MC}(\alpha)$，フェイズ α，β 間で対応付けられたメソッド呼び出しの数を $N_{match}(\alpha,\beta)$ とし，フェイズ α とフェイズ β の類似度を以下の式により定義する．

$$phase_similarity(\alpha,\beta) = \frac{N_{match}(\alpha,\beta)}{max(N_{MC}(\alpha), N_{MC}(\beta))}$$

フェイズマッチングの概要を図 4 に示す．フェイズマッチングでは，2 つの実行履歴の間のフェイズ長が閾値以上の全フェイズ間の比較を行った後，フェイズ間の類似度が高いフェイズ対からグリーディに対応付ける．なお，得られたフェイズの対を対応するフェイズと呼ぶ．

フェイズマッチングのアルゴリズムを Algorithm 2，3，4 に示す．Algorithm 2 の入力として，2 つのプログラムの実行履歴から得たフェイズの集合と，フェイズ長の閾値を与える．アルゴリズム中の Approximate-DP(Algorithm 3 の 5 行目) は，文献 [24] の Figure 9.32 の類似文字列マッチングアルゴリズムを用いたものであり，フェイズのペアを与えると，MatchPhase オブジェクトを返す．MatchPhase オブジェクトは，フェイズのペア，フェイズ間の類似度，フェイズ間のマッチング情報を持つ．sort_by_descending_similarities(Algorithm 4 の 3 行目) は，MatchPhase のリストをフェイズのペアの類似度の降順にソートする．なお，値が等しい場合は元の並びを保存する．

手順は以下の 2 つのステップからなる．

Step 1. フェイズのペアの類似度計算 (Algorithm 3) 2 つのフェイズの集合からフェイズを取り出し，共にフェイズ長が閾値以上のものに対して Approximate-DP を適用して，その結果の MatchPhase オブジェクトをリストに追加する．

Step 2. フェイズ対応付け (Algorithm 4) リスト MP_0 からフェイズのペアの類似度が高いものから順に探索していき，フェイズのペアの両方が他のフェイズとマッチしていない場合に集合 MP に追加する．

Algorithm 2 フェイズマッチングアルゴリズム

---

procedure MatchingPhases (

  in P, P': set of phases;

  in $threshold$: integer;

  out MP: set of MatchPhase);

1. list of MatchPhase $MP_0 \leftarrow$ CalcPhaseSimilarity$(P, P', threshold)$
2. set of MatchPhase $MP \leftarrow OneOnOnePhase(MP_0)$

Algorithm 3 フェイズのペアの類似度計算アルゴリズム

procedure CalcPhaseSimilarity (

  in P, P': set of phases;

  in $threshold$: integer;

  out MP: list of MatchPhase);

1. list of MatchPhase $MP \leftarrow \phi$
2. **for** p **in** P **do**
3.   **for** p' **in** P' **do**
4.   **if** $\text{len}(p) \geq threshold \wedge \text{len}(p') \geq threshold$ **then**
5.   MatchPhase $mp \leftarrow$ Approximate-DP(p, p')
6. $MP \leftarrow MP.add(mp)$
7.   **end if**
8.   **end for**
9. **end for**

Algorithm 4 フェイズ対応付けアルゴリズム

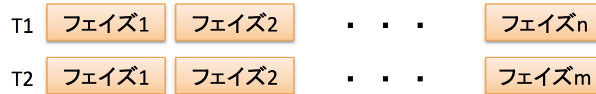
procedure OneOnOnePhase (

  in MP_0: list of MatchPhase;

  out MP: set of MatchPhase);

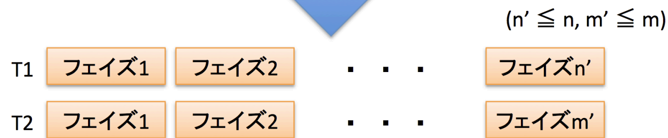
1. set of MatchPhase $MP \leftarrow \phi$
2. set of phases $Matched \leftarrow \phi$
3. $MP_0 \leftarrow$ sort_by_descending_similarities(MP_0)
4. **for** MatchPhase mp **in** MP_0 **do**
5.   **if** $\neg Matched.contains(mp.p) \wedge \neg Matched.contains(mp.p')$ **then**
6. $MP \leftarrow MP.add(mp)$
7. $Matched \leftarrow Matched.add(mp.p)$
8. $Matched \leftarrow Matched.add(mp.p')$
9.   **end if**
10. **end for**

入力: 2組のフェイズの集合



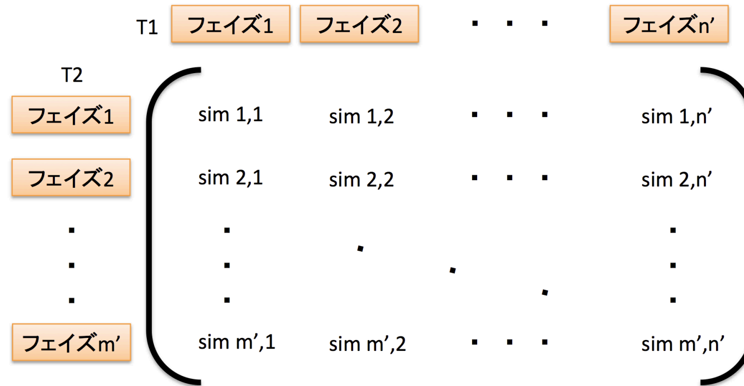
フェイズ長フィルタリング

(n' ≦ n, m' ≦ m)



類似文字列マッチング
アルゴリズム

T1 フェイズ1 フェイズ2 ・・・ フェイズn'



類似度順に対応付け

出力: 1対1に対応付けられたフェイズの組の集合 (n' ≦ m')

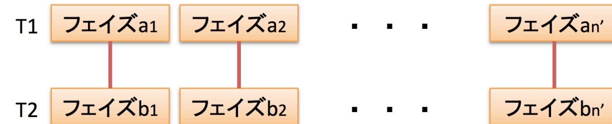


図 4: フェイズマッチング

フェイズマッチングの例を図 5 に示す．フェイズの集合 {A，B，C} と {X，Y，Z} を入力として与える．まず，フェイズ長のフィルタリングを行う．フェイズ長の閾値が 10 であるので，フェイズ長が 8 であるフェイズ Y をマッチング対象から除外する．次に，類似文字列マッチングアルゴリズムを用いて，すべてのフェイズ間の類似度を求める．最後に，フェイズ間の類似度が高いフェイズのペアから対応付ける．フェイズ C とフェイズ Z の類似度が最も高い 0.9 であるので，はじめにこの 2 つを対応付ける．次に高い類似度を持つペアはフェイズ C とフェイズ X であるが，フェイズ C はすでに対応付けられているため，これらは対応付けられない．したがって，その次に高い類似度を持つペアであるフェイズ A とフェイズ Z を対応付け，これらのペアを出力する．

3.4　クラス・メソッドマッチング

3.3 節で得られた動的なフェイズ間のメソッド呼び出しの対応関係を用いて，静的なクラス A とクラス B の類似度を以下の式により定義する．なお，クラス A 内に出現する全メソッド呼び出しの数を $N_{MC}(A)$，そしてクラス A 内とクラス B 内に現れるメソッド呼び出しのうち，対応するフェイズの対の中において対応付けられたメソッド呼び出しの数を $N_{match}(A, B)$ とする．

$$class_similarity(\mathrm{A}, \mathrm{B}) = \frac{2 \times N_{match}(A, B)}{N_{MC}(A) + N_{MC}(B)}$$

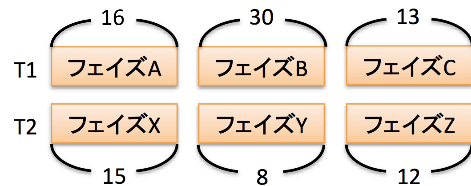
クラス間の類似度の計算例を図 6 に示す．図 6 の上部にはソースプログラムを，下部にはソースプログラムに対応する実行系列を示している．なお，ソースプログラムには，実行時に出現したメソッド呼び出しのみを記述している．また，それぞれの実行系列 α，β は 2 つのフェイズに分割されており，フェイズ α_1 とフェイズ β_1，フェイズ α_2 とフェイズ β_2 がそれぞれ対応付けられている．例えば，クラス A とクラス B について，クラス A 内のメソッド呼び出し数が 6，クラス B 内のメソッド呼び出し数が 5，クラス A 内のメソッド呼び出しとクラス B 内のメソッド呼び出しが対応付けられている数が 3 であるので，$class_similarity(A, B)$ は 0.55 となる．

また，この式のクラス A，クラス B をメソッド A，メソッド B に置き換えた式をメソッド間の類似度として定義する．

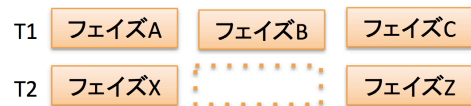
2 つの実行履歴の間の全クラス間，全メソッド間の類似度を計算し，類似度が高いものからグリーディに対応付けていき，類似クラス，類似メソッドとして出力する．

クラスマッチングのアルゴリズムを Algorithm 5，6，7，8 に示す．Algorithm 5 の入力として，3.3 節で得られた MatchPhase の集合と，クラスごとのクラス内のメソッド呼び出し数のハッシュマップを与える．アルゴリズム中の similarity(Algorithm 7 の 6 行目) は，クラ

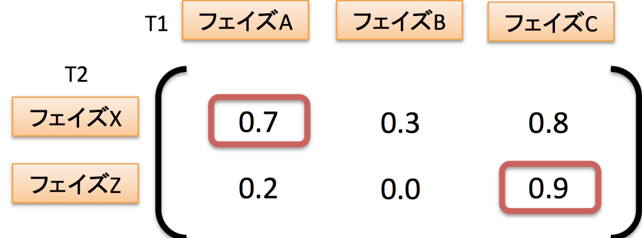
入力: 2組のフェイズの集合



フェイズ長フィルタリング



類似文字列マッチング
アルゴリズム



類似度順に対応付け

出力: 1対1に対応付けられたフェイズの組の集合

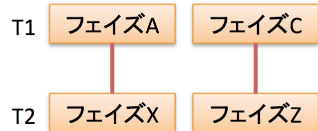


図 5: フェイズマッチングの例 (フェイズ長の閾値 10)

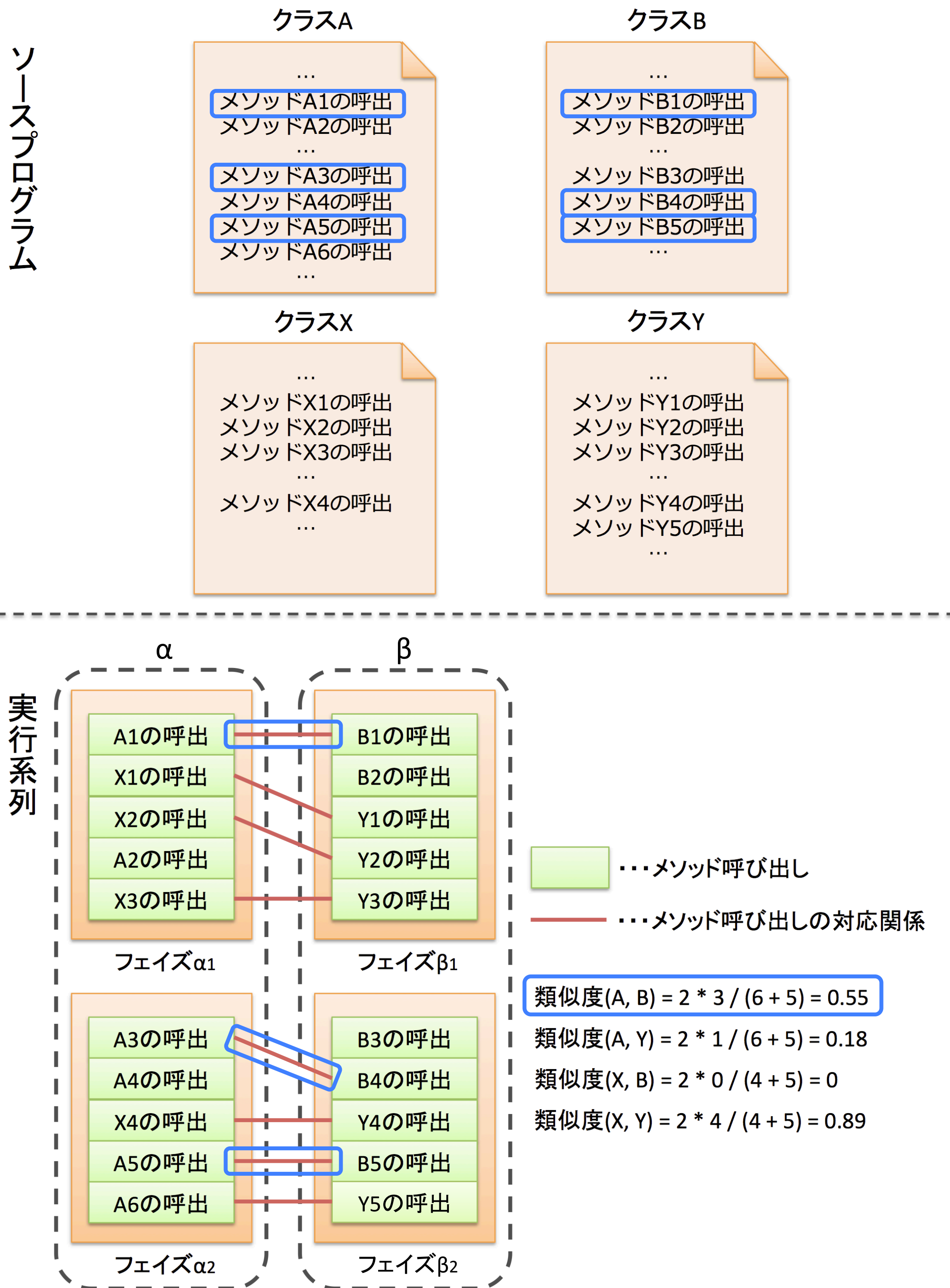


図 6: クラス間の類似度の計算例

スのペア，クラスのペアそれぞれのクラス内のメソッド呼び出し数，ペア間でマッチしているメソッド呼び出し数を与えると，MatchClass オブジェクトを返す．MatchClass オブジェクトは，クラスのペア，ペア間の類似度を持つ．sort_by_descending_similarities(Algorithm 8 の 3 行目) は，MatchClass のリストをクラスのペアの類似度の降順にソートする．なお，値が等しい場合は元の並びを保存する．

手順は以下の 3 つのステップからなる．

Step 1. **クラスのペアのマッチ数カウント (Algorithm 6)** 各ペア間でのクラス内のメソッド呼び出しのマッチ数をカウントする．このとき，各クラスは呼び出されたメソッドが定義されているクラスではなく，メソッド呼び出しが存在するクラスとする．

Step 2. **クラスのペアの類似度計算 (Algorithm 7)** すべてのクラスのペアの類似度を similarity を適用して計算し，その結果の MatchClass オブジェクトをリストに追加する．

Step 3. **クラス対応付け (Algorithm 8)** リスト MC_0 からクラスのペアの類似度が高いものから順に探索していき，クラスのペアの両方が他のクラスとマッチしていない場合に集合 MC に追加する．

クラスマッチングの例を図 7 に示す．フェイズのペアの集合 $\{<$ フェイズ α，フェイズ $\alpha'>$, $<$ フェイズ β，フェイズ $\beta'>\}$ を入力として与える．まず，フェイズ間のメソッド呼び出しの対応関係を用いて，各クラス間のマッチ数をカウントする．次に，各クラス間のマッチ数と，各クラス内のメソッド呼び出し数を用いて各クラス間の類似度を計算する．例えば，クラス A とクラス X について，クラス A 内のメソッド呼び出し数が 12，クラス X 内のメソッド呼び出し数が 20，クラス A 内のメソッド呼び出しとクラス X 内のメソッド呼び出しが対応付けられている数が 9 であるので，$class_similarity(A, X)$ は 0.56 となる．最後に，クラス間の類似度が高いクラスのペアから対応付ける．クラス A とクラス X のペア，クラス B とクラス Y のペアの類似度が高いので，これらのペアを対応付け出力する．

Algorithm 5 クラスマッチングアルゴリズム

---

procedure MatchingClasses (

　　in MP: set of MatchPhase;

　　in N: HashMap of the number of methodcall event in each class;

　　out MC: set of MatchClass);

1. HashMap $map \leftarrow CountMatchClass(MP)$
2. list of MatchClass $MC_0 \leftarrow CalcClassSimilarity(map, N)$
3. set of MatchClass $MC \leftarrow OneOnOneClass(MC_0)$

Algorithm 6 クラスのペアのマッチ数カウントアルゴリズム

procedure CountMatchClass (

in MP: set of MatchPhase;

out map: HashMap of the number of match methodcall count between class pair);

1. HashMap $map \leftarrow \phi$
2. **for** MatchPhase mp **in** MP **do**
3. **for** $< class, class' >$ **in** $mp.info$ **do**
4. **if** $map.contains(< class, class' >)$ **then**
5. $map \leftarrow map.put(< class, class' >, map.get(< class, class' >) + 1)$
6. **else**
7. $map \leftarrow map.put(< class, class' >, 1)$
8. **end if**
9. **end for**
10. **end for**

Algorithm 7 クラスのペアの類似度計算アルゴリズム

procedure CalcClassSimilarity (

in map: HashMap of the number of match methodcall count between class pair;

in N: HashMap of the number of methodcall event in each class;

out MC: list of MatchClass);

1. list of MatchClass $MC \leftarrow \phi$
2. **for** $< class, class' >$ **in** map **do**
3. integer $\#methodcall \leftarrow N.get(class)$
4. integer $\#methodcall' \leftarrow N.get(class')$
5. integer $\#matchcall \leftarrow map.get(< class, class' >)$
6. MatchClass $mc \leftarrow$ similarity($< class, class' >, < \#methodcall, \#methodcall' >, \#matchcall$)
7. $MC \leftarrow MC.add(mc)$
8. **end for**

Algorithm 8 クラス対応付けアルゴリズム

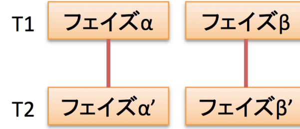
procedure OneOnOneClass (

in MC_0: list of MatchClass;

out MC: set of MatchClass);

1. set of MatchClass $MC \leftarrow \phi$
2. set of classes $Matched \leftarrow \phi$
3. $MC_0 \leftarrow$ sort_by_descending_similarities(MC_0)
4. **for** mc **in** MC_0 **do**
5. **if** $\neg Matched.contains(mc.c) \wedge \neg Matched.contains(mc.c')$ **then**
6. $MC \leftarrow MC.add(mc)$
7. $Matched \leftarrow Matched.add(mc.c)$
8. $Matched \leftarrow Matched.add(mc.c')$
9. **end if**
10. **end for**

入力: 1対1に対応付けられたフェイズの組の集合



NMC			12	9	9
		T1	クラスA	クラスB	クラスC
NMC	T2				
20	クラスX		9	2	4
14	クラスY		3	7	2

	T1	クラスA	クラスB	クラスC
T2				
クラスX		0.56	0.14	0.28
クラスY		0.23	0.61	0.17

出力: 1対1に対応付けられたクラスの組の集合

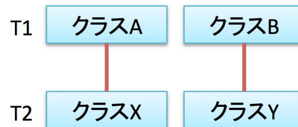


図 7: クラスマッチングの例

4　適用実験

提案手法を実装し，2 種類の適用実験を行った．1 つ目は，提案手法の難読化に対する耐性を確認するために，実際のアプリケーションに難読化を施し，難読化前後に対して手法を適用した．2 つ目は，ソースコードレベルの剽窃を特定することができるか確認するために，剽窃が疑われる 2 つの Android アプリケーションに対して手法を適用した．なお，細かい単位のフェイズを取得するために，プログラムの実行履歴のフェイズ分割におけるキャッシュサイズを 10，ウインドウサイズを 10，スコープサイズを 200，閾値を 0.01 に設定した．また，フェイズ長の閾値は 50 とした．

4.1　難読化に対する耐性の検証

提案手法の難読化に対する耐性を検証するために適用実験を行った．適用対象は統合コードクローン分析環境である ICCA [6] の Gemini コンポーネントと同じく ICCA の Virgo コンポーネントとした．この対象を選んだ理由は，ある程度の規模がありソースコードを入手できるためである．また，Gemini コンポーネントはソースコード全体のコードクローン情報を可視化することを目的として開発されており，Virgo コンポーネントはソースコードに含まれるコードクローンに関するメトリクス値を出力することを目的として開発されている．なお，Gemini コンポーネントは Java ファイル数が 120，Virgo コンポーネントは Java ファイル数が 26 である．これらコンポーネントに含まれるクラスやメソッドを対象とし，提案手法が難読化前後で同一のクラスやメソッドを識別できるかどうか確認した．

なお，難読化には，ProGuard[7]を標準設定で使用した．ProGuard は，2.4 節で述べた名前変換やメソッド分散を用いて難読化を施す．また，実行履歴の取得には Amida を使用した．

表 1: Gemini コンポーネント・Virgo コンポーネントのサイズ

	難読化前		難読化後	
	イベント数	フェイズ数	イベント数	フェイズ数
Gemini	21,560	141	21,294	92
Virgo	15,686	116	15,676	116

[6]ICCA: http://sel.ist.osaka-u.ac.jp/icca/

[7]ProGuard (http://proguard.sourceforge.net/) は，Android アプリの盗用対策に標準で用いられている．

4.1.1 クラス・メソッド単位の類似性判定

Gemini コンポーネント，Virgo コンポーネントをそれぞれ ProGuard を用いて難読化し，難読化前との類似クラスおよび類似メソッドの検出を行った．各コンポーネントの実行シナリオは，起動からコードクローンデータの解析完了までとした．取得したプログラムの実行履歴中のイベント数とフェイズ数を表 1 に示す．難読化前より難読化後のほうがイベント数，フェイズ数が少なくなっているのは，難読化を適用した結果，メソッド呼び出しのインライン展開が発生したためであると考えられる．また，検出は Xeon 2.66GHz のクアッドコアの CPU を備えた計算機上で行い，Gemini コンポーネント，Virgo コンポーネント共に検出時間は 1 秒以内であった．それぞれのコンポーネントについて，ProGuard による難読化前後のマッピング情報を用いて，類似クラス，類似メソッドの対応付けが正しいかどうかを確認した．

結果を表 2 に示す．この結果から，Gemini コンポーネントについては，ほぼすべての類似クラス，類似メソッドの対応付けが正しいことが分かる．Virgo コンポーネントについては，すべての対応付けが正しかった．これらのことから，ほぼすべてのクラスおよびメソッドを難読化前後で対応付けできたことが分かる．対応付けが失敗していたクラスやメソッドは，実行履歴中においてそれぞれから高々2 回しかメソッド呼び出しが行われていなかった．そのため，対応関係のないメソッドとの類似度が高くなってしまったメソッドや，任意のクラス (メソッド) との類似度が低くなってしまったクラス (メソッド) が存在し，正しい対応付けを行うことができなかった．

4.1.2 コードクローン検出を用いた類似性判定

トークン列の等価性に基づくコードクローン検出ツール CCFinder[9] を用いて，難読化前後のクラス，メソッドをそれぞれ正しく対応付けることができるか調査した．なお，ProGuard での難読化は jar ファイルに対して行うため，Java Decompiler[8]を用いて逆コンパイルを行い，難読化後のソースコードを取得した．

表 2: 難読化前後の類似クラス・類似メソッドのマッチング結果

	マッチング数	正解数 (提案手法)	正解数 (CCFinder)
Gemini (クラス)	61	59	24
Gemini (メソッド)	73	68	20
Virgo (クラス)	4	4	3
Virgo (メソッド)	4	4	3

[8]Java Decompiler: `http://java.decompiler.free.fr/`

結果を表 2 の右端に示す．なお，難読化前後のクラス間，メソッド間でコードクローンが 1 つでも含まれている場合に正解とした．この結果より，半数以上のクラス間，メソッド間においてクローンを検出できていないことが分かる．クラス間，メソッド間においてクローンを検出できたものについては，if-else 文の連続等の Java 言語のソースコードから頻繁に検出されるコードクローン [4] であり，同一のクラスおよびメソッドと判定するのは難しいと考えられる．また，提案手法において対応付けに失敗したクラス間，メソッド間からは，コードクローンが検出されなかった．

これらのことから，難読化したプログラムに対して CCFinder を用いた対応付けより，提案手法を用いた対応付けの方が優れていると考えられる．

4.2 剽窃が疑われるソフトウェアへの適用

剽窃が疑われる 2 つの Android アプリケーションに対して手法を適用し，これらのアプリケーションがソースコードレベルの剽窃かどうかを確認した．

4.2.1 準備

Amida では，Android アプリケーションのプログラムの実行履歴を取得することができないため，プログラムの実行履歴を出力するコードを埋め込むツールを自作した．

Android アプリケーションの実行ファイルからはクラスファイルを取得することが困難 [8] であるため，smali 形式のアセンブリコードに対してコードを埋め込んだ．Android アプリケーションの実行ファイルから smali 形式のファイルを抽出する処理は，smali[9]を使用して行った．埋め込むコードは，メソッドの開始情報と終了情報である．メソッドの開始情報は各メソッドの先頭に埋め込み，メソッドの終了情報は各 return 文の直前に埋め込む．また，メソッドの開始情報には，プログラムの実行履歴をフェイズに分割する際に必要であるオブジェクト ID を埋め込むコードも記述する．

4.2.2 関数電卓アプリケーション

RealCalc Plus[10]は，Quartic Software によって開発された関数電卓アプリケーションである．このアプリケーションは，Google Play[11]で有料で販売されている．一方で，地瓜游戲中心[12]では，このアプリケーションと同名，同機能なものが無料で配布されている．

---

[9] smali: `http://code.google.com/p/smali/`

[10] RealCalc Plus: `https://play.google.com/store/apps/details?id=uk.co.nickfines.RealCalcPlus`

[11] Google Play: `https://play.google.com/store`

[12] 地瓜游戲中心: `http://www.diguayouxi.com/`

これら2つのアプリケーションに対して，提案手法を適用した．実行シナリオは，表3に示すように，関数電卓のすべての機能を使うように設定した．取得したプログラムの実行履歴中のイベント数とフェイズ数を表4に示す．

結果を図8に示す．図8は，類似クラス，類似メソッドの類似度ごとの分布を表す．この図より，ほぼすべての類似クラス，類似メソッドの類似度が非常に高い値を示していることが分かる．

また，類似クラス，類似メソッドのマッチングが正しいかどうかを総合的に判断した．結果を表5に示す．この表より，すべての類似クラスのマッチングが正しく，ほぼすべての類似メソッドのマッチングが正しいことが分かる．

以上のことより，地瓜游戲中心で無料で配布されている RealCalc Plus は，Quartic Software によって開発されたものである可能性が高い．なお，Google Play で有料で販売されている RealCalc Plus には正規ユーザであるかの認証処理が含まれていたが，地瓜游戲中心の RealCalc Plus ではその処理が取り除かれているようであった．

表 3: 関数電卓アプリケーションの実行シナリオ

	シナリオ
RealCalc Plus	加算，減算，乗算，除算，sin，cos，tan，sinh，cosh，tanh，$\frac{1}{sin}$，$\frac{1}{cos}$，$\frac{1}{tan}$，$\frac{1}{hyp}$，小数・分数表示切り替え，x^y，$\sqrt[x]{y}$，x^2，$\sqrt{x}$，%，$\frac{1}{x}$，x^3，$\sqrt[3]{x}$，ln，e^x，log，10^x，EXP，MOD，n!，${}_nC_r$，${}_nP_r$，単位変換，π 入力，e 入力，定数入力，結果履歴表示，ゼロ除算，0!，log(0)，${}_0P_0$，${}_0C_0$，y^x，$tan(\frac{\pi}{2})$

表 4: 関数電卓アプリケーションのサイズ

	イベント数	フェイズ数
RealCalc Plus(Google Play)	76,037	99
RealCalc Plus(地瓜游戲中心)	73,260	103

表 5: 関数電卓アプリケーションの類似クラス・類似メソッドのマッチング結果

	マッチング数	正解数
類似クラス	57	57
類似メソッド	241	232

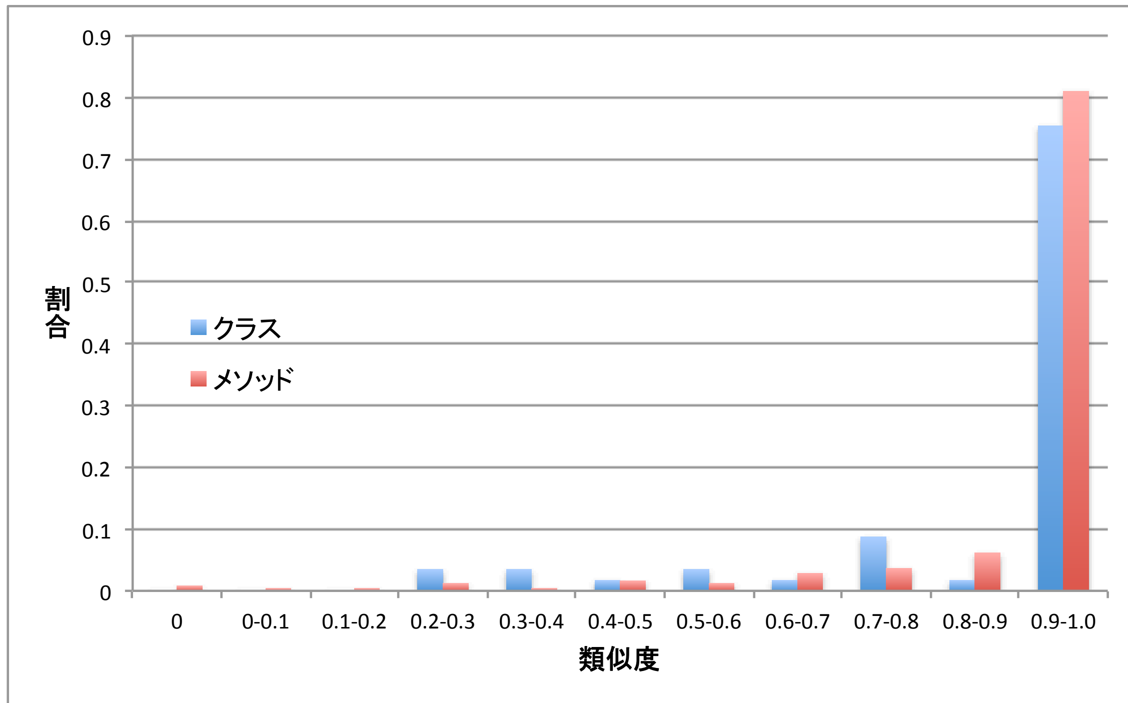


図 8: 関数電卓アプリケーションの類似クラス・類似メソッド分布

4.2.3 履歴削除アプリケーション

履歴消しゴムは，Android 端末に記録されている各種履歴を削除するアプリケーションである．この端末内の履歴を削除するアプリケーションは，DUMAPIC[13]，INFOLIFE[14] の 2 社によって開発されており，INFOLIFE が剽窃を行った疑いがあると指摘されている [3]．一方で，INFOLIFE が剽窃に関して以下のコメントを残している．

> INFOLIFE LLC 2011/07/14 21:28: Hi, I'm Eong from INFOLIFE LLC. Please pay attention that we developed the English version first. We don't know Japanese, the translation was done by 3rd party. The name of these two app may be the same. But the icon, the content and the functions are very different. Apple uses "APP STORE" to sell apps, Amazon can also use "APP STORE" for app selling. The name doesn't mean everything. You may ask why we need the internet permission, because we do collect crash information to improve the app. As a company from USA, we are trying to make better apps to make

[13]履歴消しゴム (DUMAPIC): https://play.google.com/store/apps/details?id=jp.androdev.historyeraser
[14]履歴消しゴム (INFOLIFE): https://play.google.com/store/apps/details?id=mobi.infolife.eraser

people's life easier. Please don't just denigrate without any deep investigation. (We keep the right to sue you.)

これら 2 つのアプリケーションに対して，提案手法を適用した．実行シナリオは，表 6 に示すように，各種履歴を削除することとした．取得したプログラムの実行履歴中のイベント数とフェイズ数を表 7 に示す．

結果を図 9 に示す．図 9 は，類似クラス，類似メソッドの類似度ごとの分布を表す．この図より，全体的に類似クラス，類似メソッドの類似度が低いことが分かる．

また，類似クラス，類似メソッドのマッチングが正しいかどうかを総合的に判断した．結果を表 8 に示す．この表より，すべての類似クラス，類似メソッドのマッチングが正しくないことが分かる．

以上のことより，DUMAPIC，INFOLIFE の履歴消しゴムは，独自に開発されたものであると考えられる．

表 6: 履歴削除アプリケーションの実行シナリオ

	シナリオ
履歴消しゴム (DUMAPIC)	着信履歴削除，不在着信履歴削除，発信履歴削除，よく使う連絡先削除，ブラウザ閲覧履歴削除，マーケット検索履歴削除，Gmail 検索履歴削除，アプリキャッシュ削除
履歴消しゴム (INFOLIFE)	着信履歴削除，不在着信履歴削除，発信履歴削除，よく使う連絡先削除，ブラウザ履歴削除，マーケット検索履歴削除，Gmail 検索履歴削除，クリップボード削除，アプリキャッシュ削除，発信 SMS/MMS 削除，受信 SMS/MMS 削除，SMS/MMS 下書き削除，送信失敗 SMS/MMS 削除

表 7: 履歴削除アプリケーションのサイズ

	イベント数	フェイズ数
履歴消しゴム (DUMAPIC)	1,231	5
履歴消しゴム (INFOLIFE)	1,053	5

表 8: 履歴削除アプリケーションの類似クラス・類似メソッドのマッチング結果

	マッチング数	正解数
類似クラス	14	0
類似メソッド	42	0

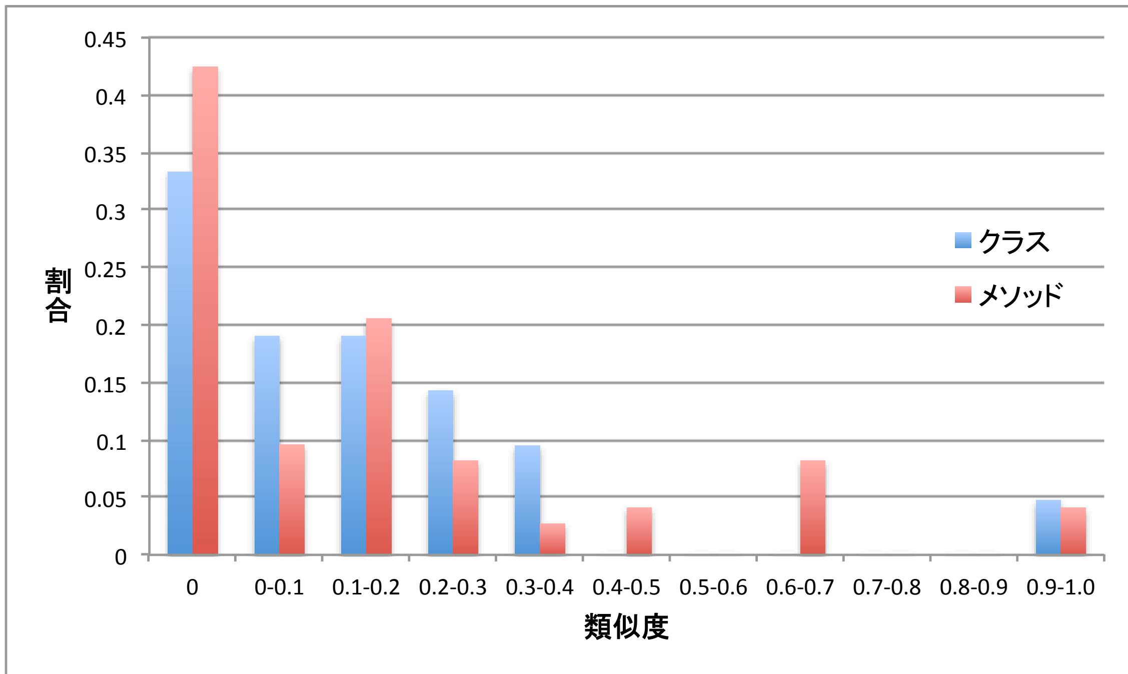


図 9: 履歴削除アプリケーションの類似クラス・類似メソッド分布

5　考察

5.1　実験結果から得られた知見

難読化に対する耐性の検証のための実験を行った．この結果より，提案手法が難読化に対する耐性を持つことが分かる．しかし，一部の出現回数の少ないクラス，メソッドのマッチングが失敗している．この問題の解決案として，マッチングに失敗したクラス，メソッドの出現回数を増やすような実行シナリオを用いることが考えられる．適切な実行シナリオを与えることによって，類似クラス間，メソッド間の類似度が高くなり，異なるクラス間，メソッド間の類似度が低くなると考えられる．

剽窃が疑われる 2 つの Android アプリケーションに対する実験では，ソースコードレベルの剽窃が行われているかを特定できることが分かった．しかし，類似アプリケーション間で低い類似度のクラスのペア (メソッドのペア) や，異なるアプリケーション間で高い類似度のクラスのペア (メソッドのペア) が存在するので，実行シナリオを追加する等を行い精度を上げる必要がある．一方で，一部のソースコードのみがコピーされた対象で実験を行うことができていないため，追加実験を行う必要がある．また，剽窃を特定する際には，提案手法の結果を根拠の 1 つとして使用し，バイナリファイルからコードクローンを検出する手法等の他のコードクローン検出ツールやバースマークの手法を併用する必要がある．

既存の静的な手法を用いて難読化前後のソフトウェアを特定することは，4.1.2 節で示したように困難であると考えられる．一方で，剽窃が疑われる Android アプリケーションを逆コンパイル[15]し，ソースコードを取得して静的な手法を適用した場合，ソースコードレベルの剽窃かどうかを特定することができると考えられる．

既存の動的な手法を用いた場合は，難読化前後のソフトウェアが同一のソフトウェアであることを確認できるが，類似クラス，類似メソッドを特定することはできないと考えられる．また，岡本らの手法が Android アプリケーションの API 呼び出しに対応できるとすると，剽窃が疑われる Android アプリケーションがソースコードレベルの剽窃かどうかを特定することができると考えられる．

5.2　妥当性への脅威

本研究で行った適用実験に関する妥当性を検証する．

本研究では渡邊らのフェイズ分割手法を用いるために，実行履歴としてメソッドの呼び出しおよびコンストラクタの呼び出しを取得した．実験で使用した Amida では他にも，フィールドの参照，定義や配列要素の定義等を取得することができるが，フェイズ分割手法では使

[15] Android アプリケーションの実行ファイルからクラスファイルを取得することは困難である [8].

用することができない．仮にフェイズ分割手法が拡張され，これらの情報によって分割の精度が向上した場合，提案手法におけるフェイズマッチングの精度が向上し，検出される類似クラス，類似メソッドの信頼性が高まると考えられる．

適用実験ではフェイズ分割におけるパラメータとして，キャッシュサイズ 10，ウインドウサイズ 10，スコープサイズ 200，閾値 0.01 を設定した．できる限り多くのフェイズを検出するために，キャッシュサイズ，ウインドウサイズを小さく設定した．一方で，スコープサイズと閾値はフェイズ分割に与える影響がキャッシュサイズ，ウインドウサイズに比べて小さいため，キャッシュサイズ，ウインドウサイズに合わせて小さめに設定した．この設定により，検出されるフェイズ数が大きくなるが，機能単位ではないフェイズも検出されてしまう．しかし，検出するフェイズの長さを 50 以上に設定することによって，検出フェイズのノイズを除去している．これらの値は手法の結果に直接影響してくるものであるため，追加実験を行って適切な値を見つけることにより，結果の精度を向上させることができると考えられる．

Android アプリケーションのプログラムの実行履歴を取得するために，プログラムの実行履歴を出力するコードを埋め込むツールを自作した．そのため，取得できるプログラムの実行履歴が Amida と自作ツールとで差異がある可能性があり，提案手法の結果に影響を及ぼす恐れがある．しかし，自作ツールは Amida のアルゴリズムを参考にして作成したため，これらのプログラムの実行履歴は同一のものとしても問題がないと考えられる．

プログラムの実行履歴出力用のコードを実験対象に埋め込むため，その影響で再描画等の余計なメソッドが呼び出される可能性がある．しかし，こういった処理はメインの処理とは別のスレッドで行われる．本研究ではスレッドごとにフェイズの分割を行なっているため，ログ出力用のコードを埋め込むことのメインの処理への影響はないと考えられる．

適用実験ではシナリオを決めて比較する 2 つのアプリケーションを実行したが，同じシナリオで実行したとしてもアプリケーションがブラックボックスであるため，バックグラウンドの処理等の差異で得られるプログラムの実行履歴が異なる可能性がある．しかし，メインの処理とバックグラウンドの処理は別のスレッドで行われるため，先ほど述べたように実験結果への影響はないと考えられる．

実験対象によっては難読化の影響が異なる可能性がある．本研究の難読化に対する耐性の評価では，ICCA の Gemini コンポーネントと Virgo コンポーネントを実験対象とした．これらのコンポーネントの Java ファイル数がそれぞれ 120，26 であり，中規模のプロジェクトと言える．そのため，難読化によるプログラムの改変が十分起こると考えられるので，難読化に対する耐性を評価するためには十分である．また，アプリケーションによっては同じシナリオを実行しても異なるプログラムの実行履歴となり，実験の結果に影響がでる可能性がある．剽窃が疑われるアプリケーションに対する実験では，関数電卓アプリケーションと

履歴削除アプリケーションを対象とした．これらのアプリケーションは実行時に乱数等を使用しないためランダム性がなく，同じシナリオを実行した際に同じ結果が得られるので，実験対象として妥当であると考えられる．

6　まとめと今後の課題

本研究では，プログラムの実行履歴をフェイズに分割し，各フェイズのメソッド呼び出し列を比較することで，類似クラス，類似メソッドを検出する手法を提案した．提案手法はプログラムの実行履歴を用いて剽窃の検出を行うため，実行ファイルのみで解析が可能である．

提案手法の有効性を確認するために，提案手法をツールとして実装し2種類の適用実験を行った．1つ目は，難読化に対する耐性を確認するために，実際のソフトウェアを難読化し提案手法を適用した．その結果，難読化前後で同一のソフトウェアを識別できることを確認した．2つ目は，剽窃の疑われるソフトウェアに対して提案手法を適用した．結果として，ソフトウェアがソースコードレベルでの剽窃であるかを識別できることを確認した．

今後の課題として，提案手法にとって適切なフェイズ分割のパラメータを得るために，パラメータごとに手法を適用して実験を行う必要がある．さらに，他の類似アプリケーションや剽窃の事例に対して実験を行い，メソッド呼び出しの正規化手法の改善を行うことが挙げられる．また，一部のソースコードを再利用したソフトウェアに対して適用実験を行い，一部のみの再利用を特定できることを示す必要がある．プログラムの実行履歴出力用のコードを埋め込んだ際の影響調査や，アプリケーションがバックグラウンドで処理を行なっている場合の影響調査を行う必要もある．最後に，プログラムの実行履歴分析を行う提案手法の有効性は実行シナリオの選択方法に依存すると考えられるため，提案手法を使用する際の実行シナリオの選択方法を考案することも今後の課題である．

謝辞

本研究において，貴重な時間を頂いて懇切丁寧なご指導及びご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感謝いたします.

本研究において，適時適切なご指導及びご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松下誠准教授に深く感謝いたします.

本研究において，日常の議論を通じて多くのご指導及びご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石尾隆助教に深く感謝いたします.

本研究において，終始丁寧に適切なご指導及びご助言を賜りました奈良先端科学技術大学院大学情報科学研究科ソフトウェア設計学講座吉田則裕助教に心より感謝いたします.

本研究において，随時適切なご指導及びご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井垣宏特任准教授に深く感謝いたします.

最後に，その他様々なご指導，ご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] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of ICSM 1998*, pp. 368–377, 1998.

[2] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, 1997. `https://researchspace.auckland.ac.nz/handle/2292/3491`.

[3] @eiKatou Blog. `http://eikatou.net/blog/2011/07/履歴消しゴムが丸パクリされた件/`.

[4] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and implementation for investigating code clones in a software system. *Information and Software Technology*, Vol. 49, No. 9-10, pp. 985–998, 2007.

[5] 肥後芳樹, 楠本真二, 井上克郎. コードクローン検出とその関連技術. 電子情報通信学会論文誌, Vol. J91-D, No. 6, pp. 1465–1481, 2008.

[6] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, Vol. 12, pp. 26–60, 1990.

[7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proc. of ICSE 2007*, pp. 96–105, 2007.

[8] JUMPERZ.NET Blog. `http://kanatoko.wordpress.com/2011/01/21/androidアプリケーションのリバースエンジニアリング/`.

[9] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, Vol. 28, No. 7, pp. 654–670, 2002.

[10] 神崎雄一郎, 門田暁人, 中村匡秀, 松本健一. ソースコードレベルにおけるプログラムのカムフラージュ. コンピュータソフトウェア, Vol. 28, No. 1, pp. 300–305, 2011.

[11] M. Kim, L. Bergman, T. Lau, and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proc. of ISESE 2004*, pp. 83–92, 2004.

[12] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. of WCRE 2001*, pp. 301–309, 2001.

[13] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *CACM*, Vol. 26, No. 6, pp. 419–429, 1983.

[14] H. Lim, H. Park, S. Choi, and T. Han. A method for detecting the theft of Java programs through analysis of the control flow information. *Information and Software Technology*, Vol. 51, No. 9, pp. 1338–1350, 2009.

[15] G. Maskeri, D. Karnam, S. A. Viswanathan, and S. Padmanabhuni. Version History Based Source Code Plagiarism Detection in Proprietary Systems. In *ICSM 2012*, pp. 609–612, 2012.

[16] 岡本圭司, 玉田春昭, 中村匡秀, 門田暁人, 松本健一. API 呼び出しを用いた動的バースマーク. 電子情報通信学会論文誌, Vol. J89-D, No. 8, pp. 1751–1763, 2006.

[17] E. Raymond and R. Landley. OSI position paper on the SCO-vs.-IBM complaint, 2004. `http://www.catb.org/~esr/hackerlore/sco-vs-ibm.html`.

[18] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proc. of ISSTA 2009*, pp. 117–128, 2009.

[19] 玉田春昭, 中村匡秀, 門田暁人, 松本健一. Java クラスファイル難読化ツール DonQuixote. 日本ソフトウェア科学会 FOSE 2006, pp. 113–118, 2006.

[20] 谷口考治, 石尾隆, 神谷年洋, 楠本真二, 井上克郎. プログラム実行履歴からの簡潔なシーケンス図の生成手法. コンピュータソフトウェア, Vol. 24, No. 3, pp. 153–169, 2007.

[21] T. Ueno. Pocketmascot に対する抗議ページ, 2001. `http://www.tomozon.sakura.ne.jp/wince/About_PocketMascot/About_PocketMascot.html`.

[22] 渡邊結, 石尾隆, 井上克郎. 協調動作するオブジェクト群の変化に基づく実行履歴の自動分割. 情報処理学会論文誌, Vol. 51, No. 12, pp. 2273–2286, 2010.

[23] R. Wettel and R. Marinescu. Archeology of code duplication: recovering duplication chanins from small duplication fragments. In *Proc. of SYNASC 2005*, pp. 63–70, 2005.

[24] R. B. Yates and B. R. Neto. *Modern Information Retrieval Second edition*. Addison Wesley, 2011.