Computer Security Symposium 2017
23 - 25 October 2017

単体テストコードを利用した動的バースマークの抽出

横井 昂典[1]　玉田 春昭[2]

概要：近年，ソフトウェアの盗用が問題となっており，盗用発見のためのバースマーク手法が注目されている．バースマーク手法は，プログラムの静的情報から抽出する静的バースマークと，動的情報から抽出する動的バースマークに分類される．動的バースマークはプログラムに入力を与え実際に動作させる必要があり，静的バースマークに比べて抽出のコストが高い．また，入力により出力されるバースマーク情報が変化するため，あらかじめ抽出しておくことが難しい．そこで，本稿では，プロジェクトの単体テストに着目し，動的バースマークの事前抽出を目指す．事前抽出が行えれば，より多くの動的バースマーク同士の比較が可能となる．評価実験では，提案手法によるバースマークの性能評価を行った．同一のプロジェクト間では類似度が 0.931 や 0.887，異なるプロジェクト間では類似度は 0.2 以下となり，良好な結果となった．また，提案手法によるバースマーク抽出のコスト評価を行った．提案手法は従来手法と比べて，入力を用意する必要がなく抽出の処理を自動化できるため，コストの削減ができているといえる．

キーワード：動的バースマーク，単体テスト，盗用発見，ソフトウェア保護

Extracting Dynamic Birthmarks From Unit Test Code

Yokoi Takanori[1]　Haruaki Tamada[2]

Abstract: Recently, software plagiarisms become serious issues, the birthmark method for plagiarism detection has been focused. The birthmark methods are classified into static and dynamic manner. The dynamic birthmark has to provide input for extracting them from program. Also, the dynamic birthmarks are hard to store, since the extracted birthmark information is sensitively changed by input. Those characteristics cause the hight cost than the static birthmark extraction. This paper focuses on unit tests in a project in order to extract birthmark beforehand. Consequently, the extraction cost of the dynamic birthmarks will decrease. In the experimental evaluations, we evaluated the performance of the birthmarks, and extraction costs. From the results, the proposed method succeeds to decrease the extraction costs of birthmark, and to equal the performance to previous method (similarity was 0.91 on average between the same project, and the different versions, and was under 0.2 among different projects).

Keywords: dynamic birthmarks, unit test, plagiarism detection, software protection

1. はじめに

ソフトウェアの盗用を発見するための技術として，ソフトウェアバースマークが提案されている [1], [2]．バースマークはソフトウェアバイナリ中の変更が困難な箇所を特徴として取り出し，互いに比較して類似性を算出する技術である．ソフトウェア中のどのような情報に着目するかによって，異なる種類のバースマークが定義されている．例えば，命令列 [3] やデータフロー [4] などである．

また，ソフトウェアの静的な情報だけでなく，実行時に得られる情報を利用した動的バースマークも提案されている [5], [6]．一般に，静的バースマークは抽出が容易であり，ソフトウェアの部品の盗用の発見のために用いられる反面，改ざんの恐れがある．また，動的バースマークは，抽出が困難で，一部の盗用発見には向かないが，改ざんに

[1] 京都産業大学大学院
Graduate School of Kyoto Sangyo University
[2] 京都産業大学
Kyoto Sangyo University

は強いと言われている．

一般にバースマーク処理は，原告（plaintiff）プログラム p が与えられた時，p と被告（defendants）プログラムの集合 $Q = \{q_1, q_2, ..., q_n\}$ からそれぞれバースマークを抽出し，p と Q のバースマークを相互に比較する．

動的バースマークの抽出が困難な理由は，大きく 2 つに分けられ，その 2 つが実行時情報を取得するためにソフトウェアを実行する必要がある点に起因する．(1) ソフトウェアを実行させるには，入力を与える必要がある．そのためには，入力を選択する必要があり，動的バースマークの抽出を困難にしている原因の一つである．(2) 一方，実行にはある程度の時間が必要である．さらに抽出の為に，プログラム中に処理のフックが行われる場合もあり，実行にさらに時間を要する．この抽出処理を比較のたびに行うと大きなオーバーヘッドとなる．

本稿では，これらの問題を解決するため，動的バースマークの事前抽出を試みる．事前抽出が行えれば，動的バースマークの抽出を困難にしている入力の選択や，実行時間の問題が解決できる．また，Q からの抽出はあらかじめ行われているため，バースマークの検査に要する時間が p からの抽出と比較時間のみになり，検査の時間の短縮につながる．

そのために，プロジェクトの単体テストに着目する．単体テストは，プロジェクトのプログラムを実行し，想定された結果が得られるかを確認するプログラムである．また，プロジェクトがビルドされるときに必ず実行され，プログラムの実行パスのほとんどを網羅するように書かれている．そのため，テストコードの実行を，動的バースマークの抽出に必要なプログラムの実行とすることで，動的バースマークの事前抽出を試みる．

2. 準備

2.1 バースマークの定義

提案手法を説明する前に，ソフトウェアバースマークの定義を説明する．ソフトウェアバースマークは玉田らによって定義されている [1], [2]．その定義を元に岡本らが動的バースマークを次のように定義した [6].

定義 1 (動的バースマーク) p, q を与えられたプログラム，I を p もしくは q に与える入力とする．$f(p, I)$ をプログラム p に入力 I を与えた時に得られる実行時情報から抽出した特徴の集合とする．このとき，以下の条件を満たすならば，$B_f(p, I) = f(p, I)$ を p の**動的バースマーク**であると言う．

条件 1 $f(p, I)$ はプログラム p に入力 I を与えることで得られる

条件 2 q が p のコピーであれば, $B(p, I) = B(q, I)$

条件 1 は，バースマークがプログラムの付加的な情報ではなく，p の実行により抽出される情報であることを示す．すなわち，バースマークは電子透かしのように付加的な情報を必要としない．条件 2 はコピーされたプログラムからは同じバースマークが得られることを示す．もしバースマーク $B(p, I)$ と $B(q, I)$ が異なっているならば，q は p のコピーではないことを意味する．ただし，入力が異なればたとえ同じプログラムであっても，異なる動的バースマークが得られる場合もある（$B(p, I) \neq B(p, J)$）．

また，バースマークは以下の保存性（Resilience, Preservation），弁別性（Credibility, Distinction）の 2 つの性質を満たすことが望まれる．

性質 1 (保存性) p から任意の等価変換により得られた p' に対して，$B(p, I) = B(p', I)$ を満たす

性質 2 (弁別性) 同じ処理を行うプログラム p と q が全く独立に実装された場合，$B(p, I) \neq B(q, I)$ を満たす．

保存性は，バースマークが様々な攻撃に対して耐性を持つことを表す．一方で，弁別性は，全く独立に作成されたプログラムは，同じ仕様であっても区別できることを示す．もちろん，この 2 つの性質を完全に満たすバースマークを提案することは困難である．そのため，実用上は，ユーザの判断により適宜強度を設定する必要がある．また，条件 1 より，プログラムに特別な情報の追加なしにバースマーク情報が構築できる点も注目すべき箇所である．

2.2 バースマークの種類

第 2.1 節に示したバースマークの定義に従い，異なる種類のバースマークが提案されている．これらは，実行時に得られる情報の構成方法や着目する情報により異なる．例えば，プログラムの実行パスに着目した手法 [5]，呼び出しメソッドに着目した手法 [6]，実行時のヒープに着目した手法 [7] などである．

一方，抽出した情報も構成方法により，異なるバースマーク情報として定義されることもある．例えば，呼び出しメソッドの系列をそのままシーケンスとして扱う EXESEQ バースマークや呼び出しメソッド系列をベクトルとして扱う EXEFREQ バースマークなどである [6].

2.3 バースマークの類似度

多くのバースマークもしくは分類で，独自の類似度計算法が定義されている．つまり，バースマーク抽出法 f で得られたバースマーク $B_f(p, I)$ と $B_f(q, I)$ の比較には，$\mathrm{sim}_f(B_f(p, I), B_f(q, I))$ が定義されている．なお，$\mathrm{sim}_f(B_f(p, I), B_f(q, I))$ の定義域は $[0, 1]$ である．

類似度が 0 であれば，両プログラムは完全に独立していることを意味し，類似度が 1 であれば，そのプログラムはコピーである疑いが非常に強いことを意味する．ただし，どの程度 1 に近ければコピーであるのかを判定するため，多くの場合，閾値 ε が導入されている．もし $B(p, I)$ と $B(q, I)$ の類似度が閾値 ε より大きい時，p もしくは q の

いずれかが他方から盗用されていることを意味する．そして，類似度の結果を以下の 3 つのグループに分類する [8].

$$\mathrm{sim}(B(p,I),B(q,I)) \begin{cases} \geq \varepsilon \\ \quad (p \text{ と } q \text{ はコピー関係にある}) \\ \leq 1-\varepsilon \\ \quad (p \text{ と } q \text{ はコピー関係ではない}) \\ \textit{otherwise} \text{ (判定不能)} \end{cases}$$

上記のように，類似度が ε 以上の場合は，プログラム p と q はコピー関係の疑いが強いことを表し，類似度が $1-\varepsilon$ 以下の場合は，コピー関係にない可能性が高いことを表す．そして，それ以外（$1-\varepsilon < \mathrm{sim}(B(p,I),B(q,I)) < \varepsilon$）の場合は，バースマークではコピー関係を判定できないことを示している．

3. キーアイデア

3.1 概要

図 1 に提案手法の概略図を示す．動的バースマーク手法は図 1 の通り，まず，(1) 原告プロジェクトと，複数の被告プロジェクトからバースマークを抽出する，続いて，(2) 原告プロジェクトのバースマークと，被告プロジェクトのバースマークを相互に比較する，

しかし，(1)，(2) ともにコストを要するため，図 1 のように大量のプログラムの比較には多大な時間と労力を要する．この問題を解決するため，本研究では (1) に着目してコストの縮小を試みる．具体的には，第 1 に，動的バースマークを事前に抽出して，結果として得られた動的バースマークを保存しておくこと，次に，単体テストを利用して動的バースマークを抽出すること，の 2 つのアプローチを採用する．

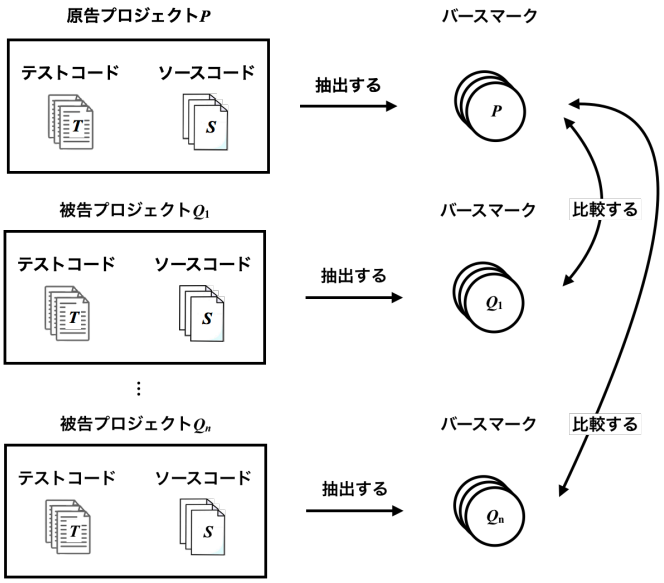

図 1 提案手法の概略図

3.2 動的バースマークの事前抽出

動的バースマークは従来から提案されているものの，プログラムの実行が必要であるため，静的バースマークに比べ抽出に時間を要する．また，プログラムを実行するためにはそのプログラムに関する知識も要する．すなわち，動的バースマークの抽出は静的バースマークの抽出に比べコストが高いといえる．

盗用を発見するバースマーク手法であるが，この手法を実行するために必要なコストが高ければ普及は難しい．普及にはやはり，より簡易に，より低コストでカジュアルに実行できることが必要である．そのためには，動的バースマークの抽出のコストを下げる必要がある．

その解決策として，動的バースマークの事前抽出を考える．従来の動的バースマークのシナリオは，比較する 2 つのソフトウェアを揃えて両者を実行して動的バースマークを抽出し，比較していた．動的バースマークは入力が変われば同じプログラムであっても出力されるバースマーク情報が異なる．そのため，両プログラムに与える入力を揃える必要がある．また，似たプログラムであれば，同じ入力を与えると同じような経路を通ると考えられるため，比較が可能となる．

一方で，あらゆる経路を通るように複数の様々な入力を与えて，複数の動的バースマークを取得し，それら全てを用いて盗用を判別できれば，事前に抽出が可能である．動的バースマークを事前に抽出できれば，あらかじめ大量のプログラムから動的バースマークを抽出しておくことが可能となる．これにより，原告プログラムから動的バースマークを抽出するだけで大量のプログラムとの動的バースマークの比較が行えるようになる．

以上のことを実現するために，単体テストに着目する．

3.3 単体テストによる動的バースマークの抽出

動的バースマークを抽出するためには与える入力を考える必要がある．与える入力によって実行パスが異なり，抽出される動的バースマークも異なるためである．しかし，あらゆる経路を通るように入力を与え，実行結果を確認する単体テストプロセスがソフトウェア開発には存在する．単体テストは，ある入力を与えたときに，結果が開発者の想定通りであるかを確認する作業である．それらの作業は JUnit[*1] や PyUnit[*2] などのテストフレームワークを通じて自動化されている．

単体テストでは，テストを行うプログラムが用意され，そのプログラム中に具体的な入力が指定されている．さらに複数の単体テストコードを用意し，プログラム全体の動作を確認する．すなわち，複数の入力が与えられ，プログラムが実行されている．

[*1] http://junit.org/
[*2] http://pyunit.sourceforge.net

一方，単体テストの十分性を計測する一つのメトリクスとして網羅率（カバレッジ）が挙げられる．テストにおいて，プログラム中の経路をどれくらい実行したかを計測するメトリクスである．網羅率はできるだけ高い方が望ましいとされている．そのため，枯れたプロジェクトであれば，網羅率は100%に近付いていると考えられる．

さらに，バースマークで行うべき検査は，特定のソフトウェア（原告）を既存の数多くのソフトウェア（被告）と比較することである．このとき，原告ソフトウェアは盗まれた可能性のあるソフトウェア，つまり，盗用か否かを判定したいソフトウェアが指定される．一方，被告ソフトウェアには自身が作成したソフトウェアを含む世の中の数多くのソフトウェアが含まれる．つまり，被告ソフトウェアには数多くのOSSが含まれることになる．OSSであれば，ソースコードも手に入り，さらに単体テストコードも含まれるプロジェクトにアクセスが可能である．

以上のことから，被告ソフトウェアから動的バースマークを抽出するために，単体テストを用いることは十分可能であり，動的バースマークの事前抽出が可能となる．なお，以降，1つのソフトウェアのソースコードやテストコードを含むものをプロジェクトと呼ぶ．

4. 提案手法

4.1 動的バースマークの自動抽出

アスペクト指向とは，横断的関心事と呼ばれる処理をプログラムの構造を横断して織り込むプログラミングパラダイムである [9]．ログ出力やログイン確認など，多くの処理で共通するものの，オブジェクト指向設計に組み込むことが難しい事項に用いられることが多い．最近では，DI (Dependency Injection) の実装として用いられている [10]．

本稿では，動的バースマークの抽出にアスペクト指向を利用する．アスペクト指向で特定のバースマークの抽出方法を定義しておき，検査したいプログラムを実行するときにアスペクトを織り込むことで動的バースマークが抽出できる仕組みを導入する．

プロジェクト P は，ソースコード集合 S，単体テストのソースコード集合 T を含む（$P = \{S, T\}$）．また，S，T はそれぞれ $S = \{s_1, s_2, ..., s_n\}, T = \{t_1, t_2, ..., t_m\}$ とする．一般に単体テストプログラムは，単体テストを行うメソッドを複数含むが，ここでは t_j は1つの単体テストのメソッドを表しているものとする．つまり，一つの単体テストプログラムに3つの単体テストを実施するメソッドがあったとすれば，単体テストコードは3つとして数える．S の各ソースコードに動的バースマーク抽出のアスペクトを織り込み，得られたソースコードを $A = \{a_1, a_2, ..., a_n\}$ とする．基本的な動作は s_i と a_i は同じであるが，a_i は実行時に動的バースマークを出力する処理が追加されているものとする．このとき，t_j を実行すると動的バースマーク $B(S, t_j)(1 \leq j \leq m)$ が得られる．なお，ここでは，t_j 内で S に与える値を入力として扱っている．

4.2 複数の動的バースマーク同士の比較

動的バースマークの定義は，$B(p, I)$ であり，プログラムと入力である．これは，入力が変われば結果となるバースマークも変わることを表す．従来の動的バースマークでは，一つの実行パスから得られるバースマーク同士を比較していた（$\mathrm{sim}(B(p, I), B(q, I))$）．提案手法では，入力を複数用意し，複数のバースマークを得ることになる．つまり，入力集合をテストコードで与える値の集合としているため，T が入力となり，$\mathcal{B}(S, T) = \{B(S, t_1), B(S, t_2), ..., B(S, t_m)\}$ が得られる．また，一般にプロジェクトが変わればテストとして与える入力も変わり，P 用の入力集合 T を異なるプロジェクト Q に与えての実行は不可能である．

以上のことから，2つのプロジェクト P と Q が与えられたとき（$P = \{S^P, T^P\}, Q = \{S^Q, T^Q\}$），バースマークの比較は $\mathcal{B}(S^P, T^P)$ と $\mathcal{B}(S^Q, T^Q)$ で行う必要がある．このとき，バースマークのペアの選択方法によって類似度が変わることは避ける必要がある．そこで，全てのバースマークのペアで比較した行列を作成し，その中から合計が最大になるようにペアを選択する [11]．得られたペアの平均類似度を両バースマークの類似度とする [12]．

行列の各要素，すなわち，従来のバースマーク同士の比較は，それぞれの動的バースマークの定義に従った類似度計算法を利用する．本手法の類似度計算は，従来手法により得られた類似度一覧から，一つの類似度を導出するものである．

5. 評価実験

5.1 概要

評価実験では，第4節で述べた提案手法に従って，動的バースマークを抽出し，比較する．以下の観点によって提案手法を評価する．

(1) バースマークの性能評価
- 弁別性評価
- 保存性評価

(2) バースマーク抽出のコスト評価

バースマークの性能評価では，提案手法による類似度計算法で，弁別性，保存性が確保できるかを確認する．続く提案手法によるコスト評価では，提案手法によりバースマーク抽出のコストが低下することを確認する．

対象とするプロジェクトは，コマンドライン引数を解析する2つのJavaライブラリ，Apache Commons CLI[*3]とArgs4j[*4]である．これらのライブラリから異なるバージョンを取得した．ここでは，バージョンの更新を盗用を隠す

[*3] http://commons.apache.org/proper/commons-cli/
[*4] http://args4j.kohsuke.org

表 1 対象プロジェクトのソースコード数とテスト数

Products	Version	$\|S\|$	$\|T\|$	C0	C1	Release
Commons CLI	1.1	20	107	83%	80%	2007
	1.3	23	364	96%	93%	2015
	1.4	23	372	96%	93%	2017
Args4j	2.0.4	20	14	47%	38%	2006
	2.0.31	62	151	77%	74%	2014
	2.33	63	162	78%	75%	2015

ための更新があったと仮定する．なお，利用したバージョンは**表 1** に示す通りである．表 1 では，利用した各バージョンのプロダクト名，バージョン番号，ソースコード数（$|S|$），テストメソッド数（$|T|$），命令網羅率（C0），分岐網羅率（C1），リリース年を示している．両プロダクト共にバージョンが上がるごとにテストが追加され，カバレッジも上昇していることがわかる．

対象とする動的バースマークは，メソッド呼び出しの頻度で表される EXEFREQ を利用した [6]．なお本稿においては，アスペクトの織り込みは手動で行なった．ただし，ビルドファイルにアスペクトを織り込むタスクを書くことで，アスペクトの織り込み作業は自動化できる．

5.2 バースマークの性能評価

ここでは，各プロダクトの各バージョンから動的バースマークを提案手法により抽出し，比較することで，動的バースマークの精度を確認する．異なるバージョンでも類似性が高ければ，保存性が確保できていることになる．また，Commons CLI と Args4j は目的は同じであるものの，全く独立に作成されたプロジェクトである．そのため，バースマークで弁別できる必要がある．つまり，異なるプロジェクト間であれば，類似性が低くなることが期待される．

また，従来の動的バースマークでは，異なるプロダクトに対しても入力を揃えて与えていた．一方で，本実験では，特に入力を揃えるといった前処理は実施していない．そのような前処理を実施しなくても，類似度を算出できることを確認する．

各プロダクトの各バージョンから抽出された動的バースマークを比較した結果を**表 2** に示す．最左列がプロダクトの各バージョンを表しており，その右列は当該プロダクトの略称を示している．続く列は行と対応しており，行，列に各プロダクトを並べ，プロダクト間のバースマークの類似度を示している．

まず，同一のプロジェクト間での比較に着目する．両プロダクト共に最新バージョンと直近の前バージョンを比較したところ，0.931(C1.3, C1.4) と 0.887(A2.0.31, A2.33) と他のペアに比べて高い類似度を示した．

一方で，同プロダクトの他のバージョン間では，類似度が Commons CLI では 0.5 以下，Args4j では 0.15 以下と非常に低い結果であった．表 1 によると，Args4j 2.0.4（最古のバージョン）は C0 = 47%，C1 = 38% と以降のバージョンに比べてカバレッジが低いことがわかる．そのため，類似度が他のバージョンと比べて低くなったと考えられる．

なお，両プロダクトの一番低いバージョンはテストコードを含む最も古いリリースバージョンである．両プロダクト共に古くからあるプロダクトであり，最新のバージョンは，最古のバージョンから 8 年以上経ってからリリースされている．

次に，バージョン間でどれくらい類似しているのかを各バージョンのソースコードで確認する．各バージョン間でソースコードが更新されているかを確認した．結果を**表 3** に示す．*from* 列が元となったバージョン，*to* 列が比較するバージョンを指す．また，$|S_c|$ がバージョン間で完全一致したソースコード数，$|S_u|$ は両バージョンに存在するが内容が更新されているソースコード数，$|S_a|$ が *from* に存在しないものの，*to* に存在するソースコード数，$|S_d|$ が *from* に存在するが，*to* に存在しないソースコード数を表す．また，*ratio* はバージョン間で完全一致したソースコードの割合であり，$\frac{|S_c|}{|S_c|+|S_u|+|S_a|+|S_d|}$ で計算される．なお，一致したか否かは `diff` コマンド[*5]を利用して調査した．

両プロダクト共に最新バージョンと直近の前バージョン間の共通のソースコードの割合は，0.870(C1.3, C1.4) と 0.810(A2.0.31, A2.33) だった．共通のソースコードが多いため，バースマークの類似度が高くなったと考えられる．

また，両プロダクト共に最古のバージョンとそれ以外のバージョン間の共通のソースコードの割合は全て 0 だった．Args4j は，Commons CLI と比較すると，バージョンの更新により追加されたソースコード数が多い．よって，Args4j の方が Commons CLI よりも類似度が低くなったと考えられる．

また，異なるプロジェクト間で比較した場合は，類似度は全て 0.2 以下となった．同一のプロジェクト間での類似度と比べると，低いことがわかる．

5.3 バースマーク抽出のコスト評価

ここでは，従来手法と，提案手法によるバースマークの抽出コストを比較する．

図 2 は提案手法と従来のバースマーク抽出のフローチャートである．それぞれのバースマーク抽出をステップに分けて示している．

まず，動的バースマークは，対象プログラムの実行時情報から抽出される．そのため，抽出には対象プログラムを実行する必要がある．提案手法では，プログラムの実行時に与える入力を用意する必要がない．しかし，従来手法での動的バースマークの抽出は，プログラムに入力を与えて

[*5] `https://www.gnu.org/software/diffutils/`

表 2　バージョンごとの動的バースマークを比較した結果

Product	Abbr.	C1.1	C1.3	C1.4	A2.0.4	A2.0.31	A2.33
Commons CLI 1.1	C1.1	1.000					
Commons CLI 1.3	C1.3	0.485	1.000				
Commons CLI 1.4	C1.4	0.485	0.931	1.000			
Args4j 2.0.4	A2.0.4	0.021	0.014	0.017	1.000		
Args4j 2.0.31	A2.0.31	0.169	0.182	0.198	0.145	1.000	
Args4j 2.33	A2.33	0.163	0.175	0.188	0.115	0.887	1.000

表 3　同一プロジェクトのバージョンの間の共通コードの割合

from	*to*	$\|S_c\|$	$\|S_u\|$	$\|S_a\|$	$\|S_d\|$	*ratio*
C1.1	C1.3	0	20	3	0	0
C1.1	C1.4	0	20	3	0	0
C1.3	C1.4	20	3	0	0	0.870
A2.0.4	A2.0.31	0	17	45	3	0
A2.0.4	A2.33	0	17	46	3	0
A2.0.31	A2.33	51	11	1	0	0.810

実行する必要がある．入力を与えて実行するということはプログラムに対しての理解が必要である．さらに，動的バースマークは与える入力によって変わるという性質がある．単一の入力では，プログラムの 1 部分のみの情報しか得られない．プログラム全体の情報を得ようとすると，複数の入力を用意する必要がある．そのため，入力を用意するにはそれなりのコストが必要となる．

一方で，提案手法による動的バースマークの抽出は，実行時に与える入力を利用者が用意する必要はない．なぜなら，プログラムの実行時に必要な入力は，テストにあらかじめ用意されているからである．また，一般にプロジェクトにはビルドファイルが用意されており，ビルドすることによりテストも実行される．そのため，ビルドファイルにアスペクトを織り込む処理を記述することで，抽出の処理を自動化できる．

これらのことから，提案手法は抽出の処理を自動化でき，入力を用意する必要もないため，提案手法は従来手法と比べると抽出のコストが低いといえる．

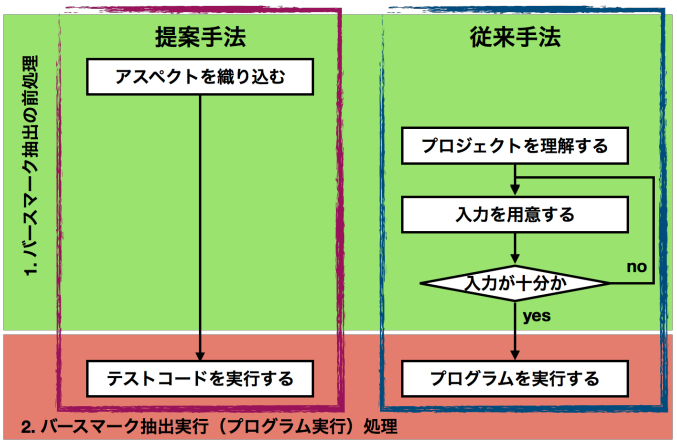

図 2　提案手法と従来手法の手順の比較

6. 利用シナリオに関する議論

バースマークは盗用を発見するものであるため，実行ファイルなどのバイナリのみから抽出できることが望ましい．しかし提案手法では，テストコードを含むソースコードを必要とする．そのため，一般的な盗用に対しての利用は難しい．

一方で，バースマークは盗用を証明するものではなく，盗用の疑いのあるプロダクトを見つけ出すものである．そのため，バースマークの本格的な利用には，大量の比較が求められる．バースマーク手法の手順は大きく抽出と比較に分けられる．

提案手法は，動的バースマークの抽出に要するコストを大きく下げ，かつ，事前の抽出を可能にする技術である．事前の抽出を可能にすることで，大量の被告プロダクトから動的バースマークをあらかじめ抽出しておくことが可能になる．そのため，比較の時には，原告プロダクトからのみ抽出し，あらかじめ抽出しておいた大量のバースマークと比較できるようになる．つまり，原告プロダクトの盗用を判定するために利用することとなる．

なお，原告プロダクトにソースコードがない場合であっても大きな問題にはならない．従来手法により，原告プロダクトから動的バースマークを抽出できるためである．その時も，入力を合わせる必要もなく，いくつかの入力を与えて抽出することで，比較が行えるようになる．

7. まとめと今後の課題

本研究では，動的バースマークの抽出のコストを削減するために，新しい抽出方法の提案を行った．プロジェクトの単体テストに着目し，テストを実行することで動的バースマークを抽出する．評価実験では，提案手法によるバースマークの性能評価とバースマーク抽出のコスト評価を行った．バースマークの性能評価では，同一のプロジェクト間では類似度が 0.931 や 0.887，異なるプロジェクト間では類似度は 0.2 以下となった．バースマーク抽出のコスト評価では，提案手法は従来手法に比べて，入力の用意の必要がなく抽出の処理を自動化できるため，抽出のコストが低いことを確認した．

今後は，より多くのプロジェクトからバースマークの抽

出を行い，バースマークの精度を確認することが課題としてあげられる．さらに，今回は抽出した動的バースマークの種類は EXEFREQ 1 種類のみであった．そのため，それ以外のバースマークに対応することも課題としてあげられる．加えて，今回の実験でのアスペクトの織り込みは手動で行った．バースマーク抽出の自動化の手順を実際に行い，その手順に要するコストの算出も必要である．

謝辞 本研究の一部は JSPS 科研費 17K00196，17K00500，17H00731 の助成を受けました．

参考文献

[1] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto. Design and evaluation of birthmarks for detecting theft of Java programs. In *Proc. IASTED International Conference on Software Engineering (IASTED SE 2004)*, pp. 569–575, February 2004.

[2] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto. Java birthmarks —detecting the software theft —. *IEICE Transactions on Information and System*, Vol. E88-D, No. 9, pp. 2148–2158, September 2005.

[3] G. Myles and C. Collberg. K-gram based software birthmarks. In *Proc. the 20th Annual ACM Symposium on Applied Computing*, pp. 314–318, 2005.

[4] Yoon-Chan Jhi, Xinran Wang, Xiaoqi Jia, Sencun Zhu, Peng Liu, and Dinghao Wu. Value-based program characterization and its application to software plagiarism detection. In *Proc. the 33rd International Conference on Software Engineering*, pp. 756–765, 2011.

[5] G. Myles and C. Collberg. Detecting software theft via whole program path birthmarks. In *Proc. Information Security the 7th International Conference, ISC 2004*, pp. 404–415, 2004.

[6] 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, August 2006.

[7] Patrick P.F. Chan, Lucas C.K. Hui, and S.M. Yiu. Jsbirth: Dynamic javascript birthmark based on the runtime heap. In *Proc. 35th Annual Computer Software and Applications Conference (COMPSAC)*, pp. 407–412, 2011.

[8] Z. Tian, Q. Zheng, T. Liu, and M. Fan. DKISB: Dynamic key instruction sequence birthmark for software plagiarism detection. In *Proc. 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC)*, pp. 619–627, November 2013.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. European Conference on Object-Oriented Programming (ECOOP 1997)*, pp. 220–242, 1997.

[10] Dhanji R. Prasanna. *Dependency Injection*. Manning Publications Co., 1st edition, 2009.

[11] Harold W Kuhn. On the origin of the hungarian method. *History of mathematical programming*, pp. 77–81, 1991.

[12] Zhenzhou Tian, Ting Liu, Qinghua ZHENG, Eryue Zhuang, Ming Fan, and Zijiang Yang. Reviving sequential program birthmarking for multithreaded software plagiarism detection. *IEEE Transactions on Software Engineering*, Vol. PP, , 2017.