情報処理学会論文誌 Vol.60 No.7 1247–1259 (July 2019)

単体テストコードとアスペクト指向を用いた動的バースマークの抽出コストの削減

横井 昂典[1] 玉田 春昭[2,a)]

受付日 2018年8月27日, 採録日 2019年4月9日

概要：盗用の疑いのあるプログラムを発見するために，プログラムの特徴を抽出・比較し類似度を算出するバースマーク手法が提案されている．バースマーク手法では，プログラム中の着目する特徴ごとに異なる手法が提案されている．本稿では特にプログラムの実行時情報を扱う動的バースマークに着目する．動的バースマークの抽出は，入力の準備とプログラムの実行が必要であることから，一般に抽出コストが高い．そこで本稿では，プログラムの単体テストを入力として扱い，また，アスペクト指向を利用し動的バースマーク抽出の自動化を目指す．これによりプログラム理解を省略でき，バースマーク処理全体に要する時間の大幅な削減が期待できる．評価実験では，提案手法を用いて抽出した動的バースマークの保存性，弁別性を評価した．異なるアーティファクトどうしの比較結果は非常に低い類似度を示し（最大 0.026，平均 0.004），同一アーティファクトの最新から 2 つのバージョンどうしの比較結果は高い類似度となり（最大 0.999，平均 0.852），提案手法は保存性・弁別性を持つといえる．次に提案手法，従来手法で抽出した動的バースマークどうしを比較した（実験 2）．比較結果は異なるアーティファクトどうしの比較では最大 0.039，平均 0.013，同一アーティファクトどうしでは最大 0.629，平均 0.235 といずれも低い類似度であった．ただし，異なるアーティファクトどうしの類似度は同一アーティファクトの結果結果に比べ，より顕著に低いため提案手法は有効であると考える．最後に，ProGuard と yGuard を用いて提案手法の攻撃耐性を評価した．実験 2 の結果との相関係数は 0.834（ProGuard），0.999（yGuard）と非常に強い相関を示した（$p < 0.0001$）．

キーワード：動的バースマーク，単体テスト，アスペクト指向，事前抽出

Reducing the Extraction Cost of Dynamic Software Birthmarks using Unit Test Codes with Aspect Oriented Programming

Takanori Yokoi[1] Haruaki Tamada[2,a)]

Received: August 27, 2018, Accepted: April 9, 2019

Abstract: The software birthmark methods calculate the similarity between two characteristics extracted from given programs for detecting the suspected copied programs. There are different birthmark method by focusing on the different characteristics of the programs. This paper focuses on the dynamic birthmarks based on runtime behaviors. To extract the dynamic birthmarks is generally costly, because, it requires preparing the inputs for the programs, and runs them. We spend much time to understand the programs for preparing inputs. The proposed method extracts the dynamic birthmarks through the running the unit test codes in a product via the aspect-oriented programming. For this, the preparing inputs phase is omitted; it is expected to reduce the cost of the birthmarking process dramatically. We conducted experiments to evaluate the resilience and credibility performance of the properties of dynamic birthmarks. At first, we compared the EXESEQ dynamic birthmarks extracted by the proposed method from 9 artifacts, and four versions each. The result showed quite low similarities among different artifacts (max 0.026, mean 0.004). Also, by comparing the latest two versions of the same artifacts, we found high similarities (max 0.999, mean 0.852). Next, we compared the EXESEQ dynamic birthmarks extracted by the proposed method and the conventional method (experiment 2). We found that the similarities among different artifacts are quite low (max 0.039, mean 0.013), and similarities between versions in the same products are relatively high (max 0.629, mean 0.235). Finally, we evaluated the tolerance against some attacks using obfuscation and optimization tools: ProGuard, and yGuard. The result showed that correlation coefficiencies are 0.8344 (ProGuard) and 0.9999 (yGuard) between the similarities from this experiment and the result from experiment 2 ($p < 0.0001$).

Keywords: dynamic birthmarks, unit tests, aspect oriented programming, beforehand extraction

[1] 京都産業大学大学院
Graduate School of Kyoto Sangyo University, Kyoto 603–8555, Japan
[2] 京都産業大学
Kyoto Sangyo University, Kyoto 603–8555, Japan
[a)] tamada@cc.kyoto-su.ac.jp

1. はじめに

ライセンスに違反してソフトウェアを利用・再利用するといったソフトウェアの盗用を発見することは，非常に困難である．世の中のどこで起こるかの予測は困難であり，さらに，実行ファイルどうしの比較は困難であるためである．実行ファイルは一般にバイナリ形式であり，人にとって解析することは困難である．そして，コンパイラのオプションやコンパイラの違いによって，バイナリの内容が大きく変わることも困難である理由の 1 つである．ソフトウェアの盗用を発見する目的のため，ソフトウェアバースマーク技術が提案された [1], [2], [3]．ソフトウェアバースマーク技術は，バイナリからそのプログラムの機能を特徴付ける部分をバースマークとして取り出す．そして，得られたバースマークどうしを比較し類似度を算出することで，元のプログラムの類似性を調べる技術である．プログラムのどの部分に着目するかにより，異なるバースマーク手法が提案されている．また，特徴の抽出方法も，プログラムを静的に解析することによる静的バースマーク [1], [3] と，プログラムを実行させたときに得られる特徴からバースマークを構成する動的バースマーク [4], [5] が提案されている．

静的バースマークはプログラムの部分の盗用検出に有用である反面，動的バースマークに比べ脆弱であることが知られている [6]．一方，動的バースマークは，プログラム全体の盗用検出に向いており，静的バースマークに比べ堅牢である反面，抽出が困難である [7], [8]．なぜなら，動的バースマークの抽出には，プログラムを実行する必要があるためである．プログラムの実行には，プログラムに入力を与える必要があり，その入力を用意するためには，プログラムへの理解が必要となる．一般に，有用な動的バースマークを抽出するには，2 つのプログラムに対して，似たような処理を行うように入力を選択する必要がある．プログラムは与える入力によって，異なる実行経路をたどるためである．このプログラムの理解が動的バースマークの抽出に要するコストの中で非常に大きなウェイトを占める．なぜならば，自動化できないためである．

前述のとおり，盗用はどこで行われるか事前に予測することは不可能である．つまり，盗用検出のためには，より多くの対象を検査する必要がある．しかし，従来の動的バースマークは，各プログラムからの抽出ごとにプログラムの理解が求められるため，検査に多大なコストを要する．そのため，現実的な問題に対応するには限界がある．

一方，静的バースマークの抽出は自動化が可能であるため，事前抽出を利用したバースマーク処理の高速化手法が提案されている [9], [10], [11]．これらは抽出されたバースマーク全体に対して，ラフで高速な比較を行い対象を絞り込むことで，全体の処理時間の短縮を図っている．これらの手法は，あらかじめ大量にプログラムを用意し，そこからバースマークを抽出しておくことが前提となっている．大量のプログラムから事前にバースマークを抽出するためには，手作業で抽出することは現実的に不可能であり，自動化が必須である．そのため，これらの手法を適用するためには，従来の動的バースマークの抽出処理のコストを大幅に下げるための自動化が必要である．

本稿では，動的バースマークの事前抽出および抽出の自動化に取り組み，バースマーク処理全体のコストを下げることを目的とする．そのために，ソフトウェアのテストコードに着目し，入力の準備のためのプログラムへの理解を不要にする．こうすることにより，動的バースマークの抽出コストを下げられ，より大規模な検査が可能となる．

以降，本稿は，2 章でバースマークの定義などを説明し，3 章で提案手法を述べる．続く 4 章で実装について説明したのち，5 章で実験の結果を報告する．最後に，6 章で本稿をまとめる．

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 [12]．

定義 1（動的バースマーク）．p，q を与えられたプログラムとし，I を p および q に与える入力とする．そしてプログラム p に入力 I を与えたときに，ある方法 f によって得られる実行時情報から抽出した特徴の集合を $B_f(p, I)$ とする．このとき，以下の条件を満たすならば，$B_f(p, I)$ を p の I における**動的バースマーク**であるという．

条件 1. $B_f(p, I)$ 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，$B_f(p, I) = B_f(q, I)$

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 $B_f(p, I)$ と $B_f(q, I)$ 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，入力が異なれば同じプログラムであっても，一般に異なる動的バースマークが得られる（$B_f(p, I) \neq B_f(p, J)$）．

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（Resilience））．p から任意の等価変換により得られた p' に対して，$B_f(p, I) = B_f(p', I)$ を満たす．

性質 2（弁別性（Credibility））．同じ処理を行うプログラム

p と q がまったく独立に実装された場合，$B_f(p,I) \neq B_f(q,I)$ 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，まった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．

2.2 バースマークの種類

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．たとえば，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 [12]，実行時のヒープに着目した手法 [6] などである．

一方，同じように抽出した情報であっても構成方法が異なれば，異なるバースマーク情報として定義される．たとえば，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の系列をベクトルとして扱う EXEFREQ バースマークなどである [12]．

これらの着目する情報や構成方法による違いを f というバースマーク抽出方法にまとめるものとする．いい換えれば，バースマーク抽出方法 f がそのままバースマークの種類であるといえる．

2.3 バースマークの類似度

多くのバースマークで，独自の類似度計算法が定義されている．つまり，バースマーク抽出方法 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，$\mathrm{sim}_f(B_f(p,I), B_f(q,I))$ の値域は $[0,1]$ 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 $B_f(p,I)$ と $B_f(q,I)$ の類似度が閾値 ε より大きいとき，p もしくは q のいずれかが他方から盗用されている疑いがあることを意味する．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 [13]．

$$\mathrm{sim}_f(B_f(p,I), B_f(q,I)) = \begin{cases} \geq \varepsilon & \text{copy relation} \\ \leq 1-\varepsilon & \text{no copy relation} \\ \text{otherwise} & \text{inconclusive} \end{cases}$$

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（$1-\varepsilon < \mathrm{sim}_f(B_f(p,I), B_f(q,I)) < \varepsilon$）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．なお，典型的な ε の値は 0.75 である [14]．

3. 提案手法

3.1 モチベーション

図 1 に動的バースマークの典型的なシナリオを示す．まず，収集段階として，動的バースマークで盗用判定を行いたいプログラムを取得する．多くの場合，盗用ではないと身元のはっきりした原告プログラムと，盗用か否かを判定したい被告プログラムである．動的バースマークの場合，原告と被告の 2 つのプログラムを比較することになる．

動的バースマークはプログラムの実行時情報から抽出するため，プログラムを動作させる必要がある．また，2.1 節で述べたように，異なる入力により異なるバースマークが抽出されるため，原告と被告の両者が同じようなバースマークになるよう入力を調整する必要がある．入力を用意するためには，両プログラムの理解が必要である．収集段階の次に行うのが，この理解段階である．

そして抽出段階にて，収集段階で取得した原告プログラム p と被告プログラム q，そして，理解段階で用意した両者に与える入力 I を元にバースマーク $B_f(p,I)$，$B_f(q,I)$ を抽出する．続く比較段階で得られた 2 つのバースマークを比較し，類似度 $s = \mathrm{sim}_f(B_f(p,I), B_f(q,I))$ を算出する．最後に，得られた s と $B_f(p,I)$，$B_f(q,I)$ を元に盗用か否かの判断を行う（検査段階）．この一連の流れがバースマークの典型的なシナリオである．

このシナリオは，動的バースマークの基本的な流れとして従来から共有されている．しかし，このシナリオの実行には多くの時間を要するという問題がある．収集，抽出，比較段階はツールを用いて自動化が可能である．しかし，

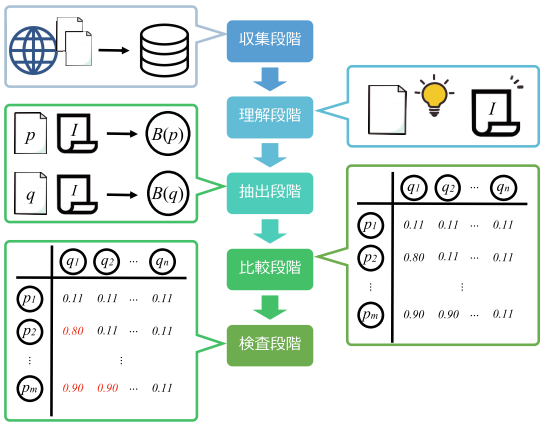


図 1 動的バースマークの典型的なシナリオ

Fig. 1 The typical scenario of the dynamic birthmarks' process.

入力を用意する理解段階は，人に依存する作業であるため自動化が困難である．そのため，バースマーク処理全体の高速化を狙うときのボトルネックとなる．

また，バースマーク手法はプログラムの盗用を発見するための手法であり，盗用を証明する手法ではないことにも注目する必要がある．盗用を発見するためには膨大な数のプログラムを調べる必要があり，対象のすべてのプログラムからバースマークを抽出しなければならない．したがって現実的には，先に述べたバースマークのシナリオを，対象を変えて何度も行う必要がある．そのたびに理解段階に多くの時間を費やしてしまうことになる．ここに，動的バースマーク処理の完全自動化が難しい理由がある．そこで本稿では，動的バースマーク処理の理解段階，抽出段階に着目し，シナリオの実行にかかる時間を短縮することを目的とする．

静的バースマークの抽出は入力の用意が不要でありプログラム単体のみから抽出できるため，動的バースマークの抽出に比べ容易である．そのため，シナリオの実行時間の短縮はバースマークの比較段階に着目して行われている [9], [10], [11], [15]．具体的には，比較処理を実施する前にラフな比較を行いまったく関係のないプログラムを除外する．これにより比較対象の大幅な削減が行える．その結果として比較時間が大幅に減少している．加えて，静的バースマークはあらかじめ抽出できるため，大量のプログラムからバースマークを抽出し，データベースなどに保存しておける．このことも全体的な処理時間の短縮に貢献している．しかし，これらの短縮方法は動的バースマークにそのまま適用できない．動的バースマークの抽出には，理解段階を経て，対象プログラムに合わせた入力を用意する必要があり，事前抽出とその自動化が困難であるためである．逆にいえば，動的バースマークの事前抽出とその自動化が実現できれば，動的バースマークにも上記の短縮法を適用できるようになり，シナリオ全体の時間短縮につながる．

3.2 動的バースマークの事前抽出

そこで本稿では，動的バースマークの事前抽出に取り組む．事前に抽出された動的バースマークはデータベースなどに保存しておくことで，比較時に参照できるようになる．

従来のバースマークのシナリオでは，大量の被告プログラムから原告プログラムに似たものを見つけ出すことが主流であった．一般に被告プログラムのソースコードが手に入るとは限らない．そのため，バースマークはバイナリを対象に特徴を抽出している．

ここで，動的バースマークのシナリオに着目し，大量の原告プログラムといくつかの被告プログラムを対象に動的バースマークを適用することを考える．原告プログラムは起源が明らかである必要があるため，プロダクトのソースコードとともにテストコードも手に入れられることが多い．そして，テストコードには一般的に，プログラムに与える入力とプログラムの動作が書かれている．いい換えると，プログラムに与える入力と，プログラムの動かし方が記載されている．そのため，テストコードが入手できれば，プログラムの理解の必要なく，入力を与え，動作させられる．つまり，テスト実行前に動的バースマークの抽出機構を注入できれば，動的バースマークが抽出できる．

なお，プロダクトには，プロダクトコードとテストコードが含まれているものとする．プロダクトコードは，そのプロダクトが提供する機能を実現するプログラム，テストコードは，プロダクトコード中のバグを発見することを目的としたテストのためのプログラムとする．一般に，テストコードは，JUnit [*1] や PyUnit [*2] などのテストフレームワークを利用し実装する．

今日では幸いなことに，開発元が明らかなソフトウェアが多数存在する．それらは，オープンソースソフトウェア（OSS）として，活発に開発・利用されている．そこで，本稿では，OSS を原告プログラムとして採用し，盗用か否かを判定したい被告プログラムを用意すれば判定できる状況を想定する．この状況設定により，原告プログラムの事前抽出が可能となり，理解段階を要するのは対象となる被告プログラムのみとなり，シナリオの大幅な時間短縮が期待できる．

3.3 単体テストによる動的バースマークの抽出

単体テストは，対象プログラムを実行してバグを発見することを目的としている．一般に，単体テストコードは対象プログラムのメソッドに具体的な入力を与え，戻り値を調べる．理想的には，単体テストコードを実行することで，対象プログラムのすべてのメソッドがテストされる．したがって，動的バースマークを抽出するための適切な入力として単体テストコードを利用できる．

テストの十分性を測定する評価指標として，カバレッジがあげられる．カバレッジはテストにより，プロダクトコードがどの程度網羅して実行されたかを表す指標であり，可能な限り高くすることが求められる．本稿では，カバレッジを動的バースマークの抽出のための入力の多様性の指標として使用する．

図 2 に提案手法の概要を示す．提案手法では，多数の原告プログラムとして OSS プロジェクトを使用している．これらの原告プログラムからは，事前に動的バースマークを抽出しておき，結果をデータベースに格納しておく．そして，被告プログラムに対しては，従来からのシナリオに従い，収集・理解段階を経て動的バースマークを抽出する．得られた被告プログラムからの動的バースマークと，デー

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

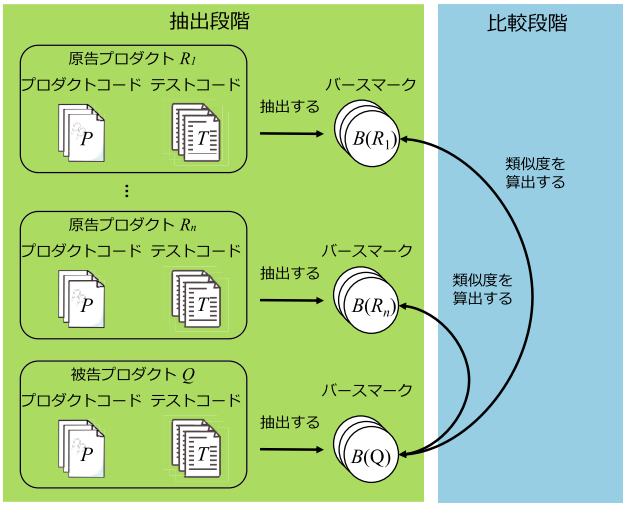


図 2 提案手法の概要図
Fig. 2 The overview of the proposed method.

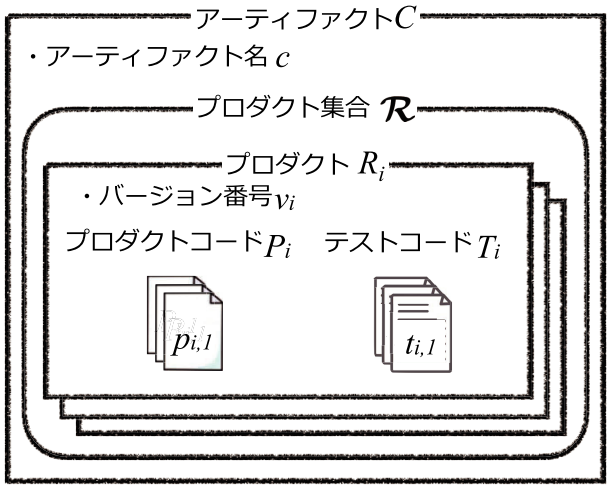


図 3 アーティファクトとプロダクトの関係性
Fig. 3 The relationships between the artifact and the product.

タベースに格納された原告プログラムからの動的バースマークを比較し，盗用の疑いがあるか否かを判定する．

3.4 動的バースマーク抽出の自動化

提案手法では，大量の原告プログラムから動的バースマークを抽出する．そのために，動的バースマークの抽出の自動化が求められる．そこで，動的バースマーク抽出機構をアスペクト指向プログラミング（AOP）を用いて構築する [16]．

AOP 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．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 [17]．

本稿ではプロダクトから動的バースマークを抽出することを考える．ここでの動的バースマーク抽出の対象となるのは，プロダクトコードである．そこで，AOP でバースマークの抽出コードをプロダクトコードに織り込み，単体テストを実行することにより，対象プロダクトから動的バースマークが抽出できるようになる．

以降，提案手法の表記について図 3 を用いて説明する．以降，アーティファクト，プロダクト，バージョンの 3 種類の分類を用いてソフトウェアを区別する．

プロダクト R_i は，バージョン番号 v_i，プロダクトコードの集合 $P_i = \{p_{i,1}, p_{i,2}, \ldots, p_{i,n_i}\}$，テストコードの集合 $T_i = \{t_{i,1}, t_{i,2}, \ldots, t_{i,m_i}\}$ を持つものとする（$R_i = \{v_i, P_i, T_i\}$）．なお，一般にテストコードには複数のテストメソッドが含まれ，各テストメソッドは単一の機能をテストするように書かれる場合が多い．そのため本稿では，1 つのテストメソッドを 1 つのテストコードとして扱う．また，アーティファクト C はアーティファクト名 c と複数のプロダクト $\mathcal{R}$ を持つ（$R_i \in \mathcal{R}$，$C = \{c, \mathcal{R}\}$）．つまり，同じ名前を持つバージョン違いのプロダクトは，同一アーティファクトとなる．

w_f を動的バースマーク B_f を抽出するためのアスペクトコードとする．次に，w_f を P_i の各プロダクトコードに織り込み，得られたコードの集合を $A_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,n_i}\}$ とする．そして，A_i がビルドされ，1 つの実行ファイル e となる．e への入力は各テストコード $t_{i,j} \in T_i$ に含まれるため，$t_{i,j}$ を入力と見なす（$1 \leq j \leq m_i$）．そして，最終的に $\mathcal{B}_f(P_i, T_i) = \{B_f(e, t_{i,1}), B_f(e, t_{i,2}), \ldots, B_f(e, t_{i,m_i})\}$ を得るものとする．この $\mathcal{B}_f(P_i, T_i)$ が，R_i から抽出した提案手法における f の動的バースマークである．

3.5 動的バースマークの比較

3.1 節で述べたように，通常，異なる入力は異なるバースマークを生成する．しかし，異なるアーティファクトに対して同一の入力を用意することは困難である．アーティファクトごとに受け入れ可能な入力の形式が一般に異なるためである．したがって，本稿では入力の統一を行わない．一方で，プロダクト R_i は一般的に複数のテストコードを持つ．すなわち $|T_i| \geq 2$ であり，すべての $t_{i,j} \in T_i (1 \leq j \leq m_i)$ を対象に動的バースマークを抽出することになる．つまり，$R_x = \{v_x, P_x, T_x\}$ と $R_y = \{v_y, P_y, T_y\}$ という 2 つのプロダクトが与えられた場合，それぞれからバースマーク f を抽出し，$\mathcal{B}_f(P_x, T_x) = \{B_f(e_x, t_{x,1}), \ldots, B_f(e_x, t_{x,m_x})\}$ と $\mathcal{B}_f(P_y, T_y) = \{B_f(e_y, t_{y,1}), \ldots, B_f(e_y, t_{y,m_y})\}$ を得る．そのとき，すべてのバースマークの各ペアを比較することによって，$m_x \times m_y$ の行列 $m_f(\mathcal{B}_f(P_x, T_x), \mathcal{B}_f(P_y, T_y))$ が生成される．なお，以下の行列では，$\mathcal{B}_f(P_i, T_i)$ の各バースマークを $b_{i,j} = B_f(e_i, t_{i,j})$ と表記している．

$$m_f(\mathcal{B}_f(P_x,T_x),\mathcal{B}_f(P_y,T_y)) = \begin{pmatrix} \mathrm{sim}_f(b_{x,1},b_{y,1}) & \cdots & \mathrm{sim}_f(b_{x,m_x},b_{y,1}) \\ \mathrm{sim}_f(b_{x,1},b_{y,2}) & \cdots & \mathrm{sim}_f(b_{x,m_x},b_{y,2}) \\ \vdots & \ddots & \vdots \\ \mathrm{sim}_f(b_{x,1},b_{y,m_y}) & \cdots & \mathrm{sim}_f(b_{x,m_x},b_{y,m_y}) \end{pmatrix}$$

次に，この行列（$m_f(\mathcal{B}_f(P_x,T_x),\mathcal{B}_f(P_y,T_y))$）をコスト行列と見なして，最大マッチングアルゴリズムを適用する [18]．すなわち，すべての可能なマッチングの中でペアの合計が最大値になるようなペアを見つける．最後に，$\mathcal{B}_f(P_x,T_x)$ と $\mathcal{B}_f(P_y,T_y)$ の類似度 s は求めたペアの平均値とする [19]．

4. 実装

提案手法に基づき，AOP を用いて動的バースマークを抽出する方法を述べる．本稿では，動的バースマークとして，EXESEQ を採用した [4], [12]．EXESEQ は，与えられたシステムライブラリのメソッド呼び出しの系列（実行順）がそのままバースマークとなる．なお，呼び出し系列はスレッドごとに記録される．本稿では，システムライブラリを Java の標準 API とする[*3]．なお，他のバースマークを抽出する場合は，異なるアスペクトコード（w_f）を開発する必要がある．

図 4 は EXESEQ を抽出するためのアスペクトコードである．提案手法では図 4 のアスペクトコードがプロダクトコードに織り込まれ，単体テストを実行することで EXESEQ の動的バースマークが抽出される．

図 4 のアスペクトコードでは，抽出されたバースマークを出力するタイミングを JVM のシャットダウン時としている．これは，EXESEQ バースマークを記録しておき，すべての単体テストが終了してから出力するためである．そのために，アスペクトコードのコンストラクタにて，JVM にシャットダウンフックを登録し，JVM 終了時にバースマークを出力するメソッドを呼び出すようにしている．

アスペクトを織り込むポイントカットは，`@Pointcut` アノテーションがつけられた 3 つのメソッドである．1 つ目は `runAspect` メソッドであり，`ExeSeqExtractor` のメソッドが実行される場合に処理が行われる．2 つ目は `callAPI` メソッドで，`java` から始まるパッケージに所属するクラスのメソッドが呼び出されるときに処理が実行される．最後に，`execTest` メソッドは，メソッドの名前が `test` から始まるメソッドが実行される前に処理が行われる．

次に，`execTest` に該当するが `runAspect` に該当しないポイントカットについて，次の処理を行うようにする．`beforeTest` メソッドは，`execTest` の処理の前に実行されテストメソッド名を格納する．そして，`afterTest` メソッドはテストメソッドの呼び出しが終わったときに実行され，出力フォーマットを整えて `results` に保存する．そして，実際に動的バースマークの要素を記録するメソッドが `storeElement` メソッドである．このメソッドでは，`callAPI` の処理の前に呼び出され，呼び出されたメソッドを記録する．図 4 の `findList` から分かるように，EXESEQ の定義より，バースマークの要素はスレッド ID ごとに収集している．

[*3] 簡単化のため本稿では，`java`，`javax` で始まるパッケージのみをシステムライブラリとする．Java の標準 API には他のパッケージ（たとえば，`org.w3c.dom` など）が含まれているが省略する．

```
// The import statements are omitted.
@Aspect
public class ExeSeqExtractor{
  Map<Long, List<String>> birthmarks
      = new TreeMap<>();
  List<String> results = new ArrayList<>();
  public ExeSeqExtractor() {
    Runtime.getRuntime().addShutdownHook(
        new Thread(() -> output()));
  }
  @Pointcut("within(ExeSeqExtractor)")
  public void runAspect() {}
  @Pointcut("call(* java..*(..))")
  public void callAPI () {}
  @Pointcut("execution(* * ..*test*(..))")
  public void execTest() {}
  @Before("callAPI() && !cflow(runAspect())")
  public void storeElement(JoinPoint jp) {
    addBirthmark(jp.toString());
  }
  @Before("execTest() && !cflow(runAspect())")
  public void beforeTest(JoinPoint jp) {
    if(!jp.toString().matches(".*junit.*"))
      results.add(jp.toString());
  }
  @After("execTest() && !cflow(runAspect())")
  public void afterTest(JoinPoint jp) {
    if(!jp.toString().matches(".*junit.*"))
      formatForOutput();
    birthmarks.clear();
  }
  public List<String> findList(long threadId) {
    List<String> list = birthmarks.getOrDefault(
        threadId, new ArrayList<>());
    birthmarks.put(threadId, list);
    return list;
  }
  public void addBirthmark(String calledAPI) {
    long threadId = Thread.currentThread().getId();
    findList(threadId).add(calledAPI);
  }
  public void formatForOutput() {
    results.add(birthmarks.entrySet().stream()
      .map(e -> toString(e.getKey(),e.getValue()))
      .collect(Collectors.joining(", ")));
  }
  private String toString(
        long id, List<String> list) {
    return String.format("%d %s",
        id, toString(list));
  }
  private String toString(List<String> list) {
    return list.stream()
          .collect(Collectors.joining(" "));
  }
  public void output() {
    results.stream().forEach(System.out::println);
  }
}
```

図 4 EXESEQ バースマークの抽出のためのアスペクトコード
Fig. 4 The aspect code for extracting EXESEQ dynamic birthmarks.

5. 評価実験

5.1 実験概要

本節では提案手法を (a) 有効性，(b) 従来手法との比較，(c) 攻撃耐性，そして，(d) 抽出コストの 4 つの観点から評

価する．まず (a) 有効性では，バースマークの性質である保存性と弁別性を満たすかを確認する．具体的には，いくつかのカテゴリを選択し，そのカテゴリごとに複数のバージョンのプロダクトを含むアーティファクトを複数用意する．そしてすべてのプロダクトから提案手法を用いて動的バースマークを抽出し，相互に比較する．そして，あるアーティファクトにおいて，あるバージョン i を基にして別バージョン j が開発されたならば，バースマークの観点では j は i を盗用したと見なせる．したがって，同じアーティファクト，異なるバージョンのプロダクト間の比較では高い類似度が示されるはずである．一方，異なるアーティファクト間の類似度はたとえ同じカテゴリであっても低くなるはずである．

次に (b) 従来手法との比較では，提案手法で抽出した動的バースマークと従来手法で抽出した動的バースマークの比較を行う．これにより，提案手法の利用シナリオの有効性を確認する．(c) 攻撃耐性では，難読化・最適化ツールを使いプロダクトを変換する．そして，その前後でバースマークを比較し類似度の変化を評価する．最後に (d) 抽出コストでは，提案手法と従来手法の抽出コストを定性的に評価する．

なお，EXESEQ の抽出には，AspectJ 1.8.10 [*4] を利用し，4 章で述べたアスペクトコードを用いた．図 4 を対象のプロダクトコードに織り込み，プロダクトのテストコードを JUnit 5 上で実行することで EXESEQ を抽出した．

5.2 対象プロジェクト

本評価実験では，OSS プロジェクトから抽出された動的バースマークを比較する．世の中の OSS プロジェクトから，3 つのカテゴリ，3 つのアーティファクト，および 4 つのバージョンのプロダクト（3 カテゴリ × 3 アーティファクト × 4 バージョン = 36 プロダクト）を選択する．カテゴリはコマンドライン，JSON，および CSV のライブラリである．各カテゴリ，およびアーティファクトは人気順の上位 3 アーティファクトとした．

表 1 に利用したプロダクトとそのメトリクスを示す．表 1 の列は左列から順に，カテゴリ（Cate.），アーティファクト名（Artifacts），バージョン（Version）を表している．続いて，$|P|$，$|T|$，C0，C1 および Year は，プロダクトコードのファイル数，テストメソッドの総数，ステートメントカバレッジ（命令網羅率；C0），ブランチカバレッジ（分岐網羅率；C1），およびリリース年である．ほとんどのプロダクトのカバレッジはバージョンの更新によって増加していることが分かる．

5.3 提案手法の有効性評価

この実験では，2.1 節に示す動的バースマークの保存性と弁別性の観点から提案手法の有効性を評価する．比較結果のヒートマップを**図 5** に示す．縦軸にアーティファクトを並べ，各アーティファクトに表 1 に示したバージョンを古い順に上から並べている．横軸も縦軸と同じアーティファクト，バージョンを並べている．なお，アーティファクトごとに白の補助線を引いている．そして右上に，類似度が 0 から 1 に対応する色を示している．

対角線に赤が並んでいるのは，同一アーティファクトかつ同一バージョンでの比較であり，類似度が 1.0 であるた

表 1 利用したプロダクトのメトリクス
Table 1 The metrics of the target products.

Cate.	Artifacts	Version	$\|P\|$	$\|T\|$	C0	C1	Year
Command Line Interface	Commons CLI[*5]	1.1	20	214	83%	80%	2007
		1.2	20	187	96%	91%	2010
		1.3	22	364	96%	93%	2015
		1.4	22	372	96%	93%	2017
	Args4j[*6]	2.0.8	24	33	67%	60%	2008
		2.0.16	40	59	73%	67%	2009
		2.0.31	62	151	76%	73%	2014
		2.33	63	162	77%	74%	2015
	picocli[*7]	1.0.0	2	343	93%	87%	2017
		2.0.0	2	406	92%	85%	2017
		3.2.0	2	899	92%	87%	2018
		3.3.0	2	904	92%	87%	2018
JSON	Gson[*8]	1.1	74	204	68%	60%	2008
		2.4	62	966	84%	79%	2015
		2.8.0	63	1,014	83%	79%	2016
		2.8.2	63	1,014	83%	79%	2017
	Jettison[*9]	1.0	27	49	49%	42%	2008
		1.3.1	37	80	48%	40%	2011
		1.3.7	38	115	54%	47%	2014
		1.3.8	38	119	54%	47%	2016
	Flexjson[*10]	3.0	74	95	69%	68%	2013
		3.1	74	96	69%	68%	2013
		3.2	74	99	69%	68%	2014
		3.3	74	103	69%	69%	2014
CSV	Nuiton CSV[*11]	3.0-rc-1	34	12	45%	42%	2014
		3.0-rc-2	34	13	45%	42%	2014
		3.0-rc-4	34	15	47%	43%	2014
		3.0-rc-5	34	15	46%	42%	2015
	JCSV[*12]	1.2.0	24	24	81%	73%	2012
		1.3.0	35	42	65%	67%	2012
		1.3.2	35	42	65%	67%	2012
		1.4.0	36	43	63%	67%	2012
	CSV Stream[*13]	0.0.1	2	25	86%	62%	2016
		0.0.2	2	33	87%	70%	2017
		0.0.4	2	44	86%	73%	2017
		0.0.5	5	63	71%	56%	2018

[*4] https://www.eclipse.org/aspectj/

[*5] https://commons.apache.org/proper/commons-cli/
[*6] http://args4j.kohsuke.org/
[*7] https://picocli.info/
[*8] https://github.com/google/gson
[*9] https://github.com/jettison-json/jettison
[*10] http://flexjson.sourceforge.net/
[*11] http://nuiton-csv.nuiton.org/v/latest/
[*12] https://code.google.com/archive/p/jcsv/
[*13] https://github.com/ansell/csvstream/

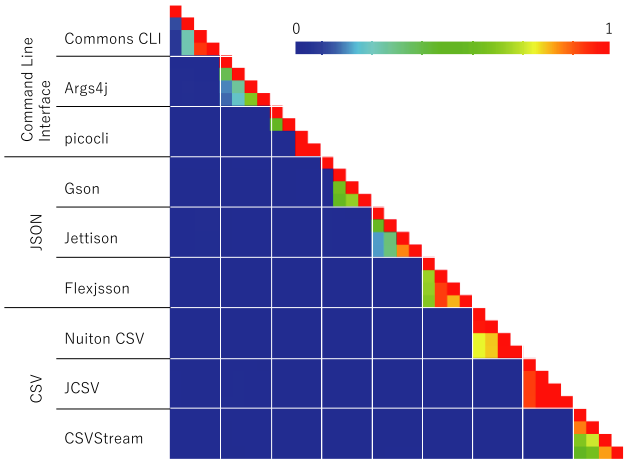


図 5 プロダクト間の比較結果

Fig. 5 The comparison results among the products.

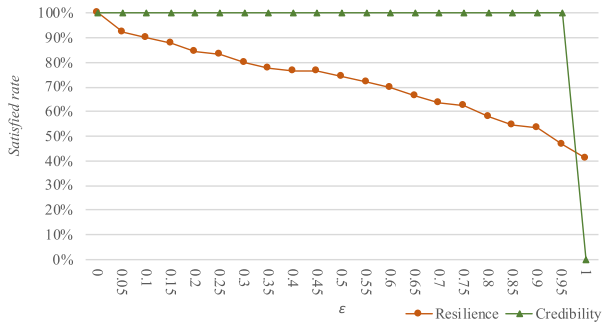


図 6 保存性，弁別性を満たす割合（提案手法での比較）

Fig. 6 Satisfied rate of resilience and credibility (Comparing by the proposed method).

めである．図 5 から，提案手法は，異なるアーティファクトのプロダクトどうしであれば，最大値 0.026，最小値 0.000，平均値 0.004，中央値 0.003 と非常に低い類似度となり，弁別性を満たしているといえる．

一方，同じアーティファクト，異なるバージョン，すなわち，対角線付近を見てみると，バージョンの差が大きいと類似度が低くなる．一方，バージョンの差が小さいと高い類似度を示す傾向があることが分かる．

次に，保存性（Resilience），弁別性（Credibility）を満たした割合を**図 6** に示す．横軸が ε の値，縦軸が満たした割合を表す．弁別性の実線は，異なるアーティファクトのプロダクトどうしを比較した結果のうち，$\mathcal{B}_f(P_x, T_x)$ と $\mathcal{B}_f(P_y, T_y)$ の類似度 s が $1-\varepsilon$ より小さくなったプロダクトが，全体のうちどれくらいを占めるかを表す．保存性の実線は，同じアーティファクト内で相互に比較したとき，類似度 s が ε より大きくなったプロダクトの割合を表す．

図 6 の保存性の評価には，同アーティファクト，同バージョンの比較も含まれている．そのため，$\varepsilon = 1$ であっても，37 ペアが保存性を満たす結果となっている．なお，$\varepsilon = 1$ のときに類似度が 1 になったのは，同バージョンの比較を除くと 1 ペアのみ，JCSV の 1.3.0 と 1.3.2 であった．

図 6 から，保存性は ε が増加するごとに満たす割合が下がっていく．同アーティファクト内の比較は，理想的には保存性を満たす，すなわち，高い類似度が望まれる．しかし，変更箇所が多いと，バースマークも大きく変わることがある．そのため，類似度が低くなると考えられる．

次に，同じアーティファクト内のプロダクトどうしの違いを調査する．**表 2** にソースコードを元にした比較結果と提案手法による比較結果の類似度を示す．各列には示されたアーティファクト名の *from* と *to* のバージョンが比較される．P_c，P_u，P_a，P_d は完全一致したプロダクトコードのファイル数，変更された数，追加された数，削除された数を示している[*14]．追加と削除は，ファイルが *to* または *from* にのみ存在することを意味する．*ratio* は，$\frac{P_c}{P_c+P_u+P_a+P_d}$ により算出され，バージョン間の完全一致したプロダクトコードの割合（一致率）である．そして，最後の列の s は，提案手法による類似度である．

最新および 1 つ前のバージョン間の *ratio* は，CSVStream 以外のすべてのプロダクト間で 0.8 を超えている．ただし，表 1 より，CSVStream は $|P| = 5$ とプロダクトコードの数が少ないため，少しのファイルの追加が *ratio* の大きな低下につながっていると考えられる．

なお，*ratio* と類似度 s の相関係数は 0.882 と非常に大きな値となっている．これは，ソースコードが似ていれば，提案手法による類似度も高くなることを表している．すなわち，提案手法はバースマークとして有効であるといえる．

5.4 提案手法と従来手法との比較

提案手法では動的バースマークの事前抽出を可能とすることで，抽出コストの削減を行う．一方で，提案手法の利用シナリオに立ち返って考えると，被告プログラムからは従来手法を用いて動的バースマークを抽出する必要がある．なぜなら，基本的に被告プログラムからはソースコードが手に入らないためである．そのため，テストコードの入手が不可能であり，提案手法は適用できない．このとき問題になるのが，従来手法，提案手法，それぞれで抽出したバースマークを比較する必要があることである．本実験では，従来手法，提案手法，それぞれで抽出したバースマークを比較し，保存性と弁別性を満たすかを確認する．実験の対象は，表 1 に示した 9 つのアーティファクトとそれぞれの 4 バージョンである．

ただし前提として，従来手法で得られるバースマークの系列は基本的には 1 つであり，提案手法で得られるバースマークの系列は複数個である．そこで，従来手法で得られたバースマークを $L(P_i, I)$，提案手法のバースマークを $\mathcal{B}_f(P_i, T_i) = \{B_f(P_i, t_{i,1}), \ldots, B_f(P_i, t_{i,m_i})\}$ としたとき，類似度を次の手順で求める．まず，$L(P_i, I)$ と各 $B_f(P_i, t_{i,j})$

[*14] これらは `diff` コマンドによって確認した（https://www.gnu.org/software/diffutils/）

表 2 同一アーティファクト内のソースコードの類似性評価
Table 2 The similarities among the products in the same artifact from the source codes.

	from	*to*	P_c	P_u	P_a	P_d	*ratio*	*s*
Commons CLI	1.1	1.2	2	18	0	0	0.100	0.103
	1.1	1.3	0	20	2	0	0.000	0.065
	1.1	1.4	0	20	2	0	0.000	0.065
	1.2	1.3	0	20	2	0	0.000	0.264
	1.2	1.4	0	20	2	0	0.000	0.264
	1.3	1.4	18	4	0	0	0.818	0.915
Args4j	2.0.8	2.0.16	10	14	16	0	0.250	0.387
	2.0.8	2.0.31	1	23	38	0	0.016	0.156
	2.0.8	2.33	1	23	39	0	0.016	0.137
	2.0.16	2.0.31	7	32	23	1	0.111	0.292
	2.0.16	2.33	7	32	24	1	0.109	0.221
	2.0.31	2.33	51	11	1	0	0.810	0.629
picocli	1.0.0	2.0.0	1	1	0	0	0.500	0.468
	1.0.0	3.2.0	0	2	0	0	0.000	0.014
	1.0.0	3.3.0	0	2	0	0	0.000	0.014
	2.0.0	3.2.0	0	2	0	0	0.000	0.013
	2.0.0	3.3.0	0	2	0	0	0.000	0.013
	3.2.0	3.3.0	2	0	0	0	1.000	0.995
Gson	1.1	2.4	0	23	39	51	0.000	0.010
	1.1	2.8.0	0	23	40	51	0.000	0.010
	1.1	2.8.2	0	23	40	51	0.000	0.008
	2.4	2.8.0	25	37	1	0	0.397	0.593
	2.4	2.8.2	22	40	1	0	0.349	0.542
	2.8.0	2.8.2	51	12	0	0	0.810	0.661
Jettison	1.0	1.3.1	9	17	11	1	0.237	0.545
	1.0	1.3.7	5	21	12	1	0.128	0.174
	1.0	1.3.8	5	21	12	1	0.128	0.173
	1.3.1	1.3.7	18	19	1	0	0.474	0.338
	1.3.1	1.3.8	18	19	1	0	0.474	0.334
	1.3.7	1.3.8	31	7	0	0	0.816	0.841
Flexjson	3.0	3.1	69	5	0	0	0.932	0.671
	3.0	3.2	67	7	0	0	0.905	0.659
	3.0	3.3	56	18	0	0	0.757	0.635
	3.1	3.2	67	7	0	0	0.905	0.908
	3.1	3.3	56	18	0	0	0.757	0.908
	3.2	3.3	74	0	0	0	1.000	0.806
Nuiton CSV	3.0-rc-1	3.0-rc-2	32	2	0	0	0.941	0.962
	3.0-rc-1	3.0-rc-4	29	5	0	0	0.853	0.757
	3.0-rc-1	3.0-rc-5	28	6	0	0	0.824	0.756
	3.0-rc-2	3.0-rc-4	30	4	0	0	0.882	0.792
	3.0-rc-2	3.0-rc-5	30	4	0	0	0.882	0.790
	3.0-rc-4	3.0-rc-5	32	2	0	0	0.941	0.996
JCSV	1.2.0	1.3.0	21	1	13	2	0.568	0.907
	1.2.0	1.3.2	21	1	13	2	0.568	0.907
	1.2.0	1.4.0	21	1	14	2	0.553	0.906
	1.3.0	1.3.2	35	0	0	0	1.000	1.000
	1.3.0	1.4.0	34	1	1	0	0.944	0.999
	1.3.2	1.4.0	34	1	1	0	0.944	0.999
CSVStream	0.0.1	0.0.2	1	1	0	0	0.500	0.850
	0.0.1	0.0.4	1	1	0	0	0.500	0.629
	0.0.1	0.0.5	1	1	3	0	0.200	0.497
	0.0.2	0.0.4	1	1	0	0	0.500	0.729
	0.0.2	0.0.5	1	1	3	0	0.200	0.599
	0.0.4	0.0.5	1	1	3	0	0.200	0.823

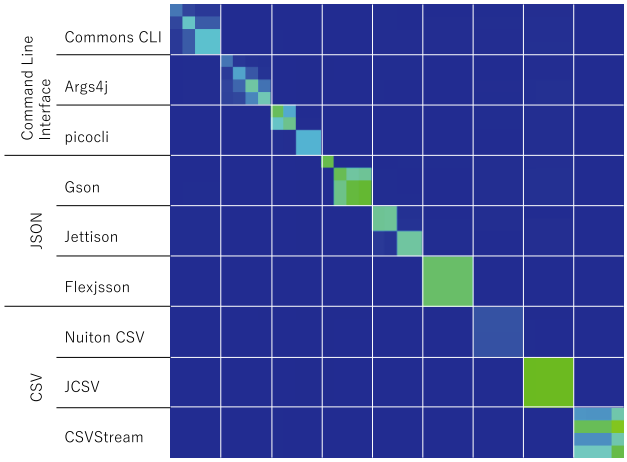


図 7 提案手法と従来手法それぞれで抽出したバースマークの比較
Fig. 7 The comparison results between two birthmarks extracted from the conventional method and the proposed method.

を比較し，類似度の系列 $\mathcal{S}(L(P_i,I),\mathcal{B}_f(P_i,T_i)) = \{s_j | s_j = \mathrm{sim}(L(P_i,T_i),B_f(P_i,t_{i,j}))\}(1 \leq j \leq m_i)$ を得る．次に，$\mathcal{S}(L(P_i,I),\mathcal{B}_f(P_i,T_i))$ の中から最大値となる s_j を取り出し，これを両者の類似度とする．

なお，提案手法で得られる動的バースマークは単機能の実行の振舞いであり，従来手法では機能での絞り込みは行われていない．さらに，一般に従来手法で得られる動的バースマークの方が系列は長くなると予測できる．そのため，この方法で得られる類似度は，一般的なバースマークの比較によって得られる類似度と比べ，低くなると予測できる．そこで，類似プロダクトと非類似プロダクトの類似度の結果から閾値を求めるものとする．

比較結果を類似度のヒートマップとして**図 7** に示す．縦軸に各アーティファクトから提案手法によって抽出したバースマークを並べる．そして，横軸にも同じ順序でアーティファクトを並べ，各アーティファクトから従来手法で抽出したバースマークを示している．なお，図 5 と同様，アーティファクトごとに白の補助線を引いている．そして，対応するバースマークどうしの比較結果である類似度を示している．図 7 から分かるように，異なるアーティファクトのプロダクト間の類似度は非常に小さい．実際の値を見てみると，最大値 0.039，最小値 0.002，平均値 0.013，中央値 0.011 であり，まったく類似していないことが分かる．

図 8 に保存性（Resilience），弁別性（Credibility）を満たした割合のグラフを示す．横軸が ε の値，縦軸が満たした割合である．図 8 から，$\varepsilon = 1$ 以外はすべてのペアが弁別性を満たしている．一方，保存性は $\varepsilon = 0$ から急激に割合が下がり，$\varepsilon = 0.6$ のときに 1%，それ以降，0%となった．

同じアーティファクトのプロダクト間の類似度を確認すると，最大値 0.629，最小値 0.010，平均値 0.235，中央値 0.190 と一般的なバースマークの類似度としては低い値に

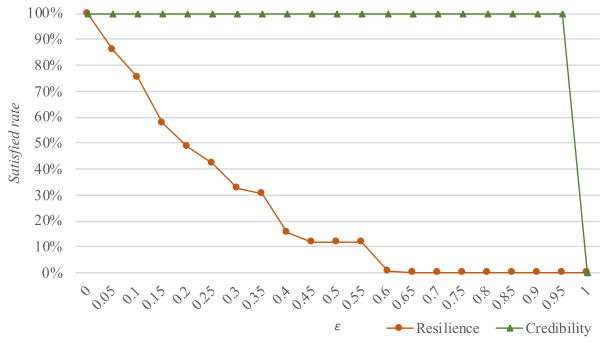


図 8 保存性，弁別性の満たす割合（提案手法と従来手法の比較）
Fig. 8 Satisfied rate of resilience and credibility (Comparing by the proposed method and the conventional method).

なっている．しかし，前述のとおり低い類似度になることが見込まれる．そして，異なるアーティファクト間の類似度は最大でも 0.039，平均 0.013 と非常に小さい．そのため，閾値 ε を 0.1〜0.3 程度にすると盗用されたかを判定できると考えられる．なお，$\varepsilon = 0.1$ としたときに判定できた割合は 75%（= 108/144 [*15]）であった．同様に，$\varepsilon = 0.2$ のとき，49%（70/144），$\varepsilon = 0.3$ のとき，33%（47/144）となり，当然ながら閾値を上げるごとに判定できる割合は下がる．バースマークは疑いのあるものを見つけ出す手段であるため，一般にバースマークの検査に続いて別の手段の検査が行われる．そのため，検出漏れを起こさない程度に閾値を下げることが望ましい．具体的な閾値は，検査結果を元にユーザがつど判断する必要があるといえる．

5.5 攻撃耐性の評価

ここでは，プログラムの盗用者が盗用の事実を隠すために，何らかの等価変換を行った状況を想定し，その状況での提案手法の有効性を評価する．そのために，原告プログラムから提案手法を用いて抽出したバースマークと，被告プログラムから従来手法を用いて抽出したバースマークを比較する．なお，被告プログラムは難読化・最適化ツールを用いて変換されたものを用いる．

本実験で等価変換に用いたツールは ProGuard [*16] と yGuard [*17] である．両ツールともに難読化および最適化を行うツールである．

図 9，**図 10** はそれぞれ ProGuard，yGuard を用い被告プロダクトと原告プロダクトを比較した類似度の一覧をヒートマップとして表した図である．それぞれ，縦軸に原告プロダクトを，横軸に同じ順序で被告プロダクトを並べている．なお，図 7 と同様に，アーティファクトごとに白の補助線を引いている．図 9 を図 7 と比較すると，JCSV 以外を除いてほぼ同じであることが分かる．しかし，JCSV どうしの比較結果を具体的な数値で見ると，たとえ

[*15] 4 バージョン × 4 バージョン × 9 アーティファクト = 144
[*16] https://www.guardsquare.com/en/products/proguard
[*17] https://www.yworks.com/products/yguard

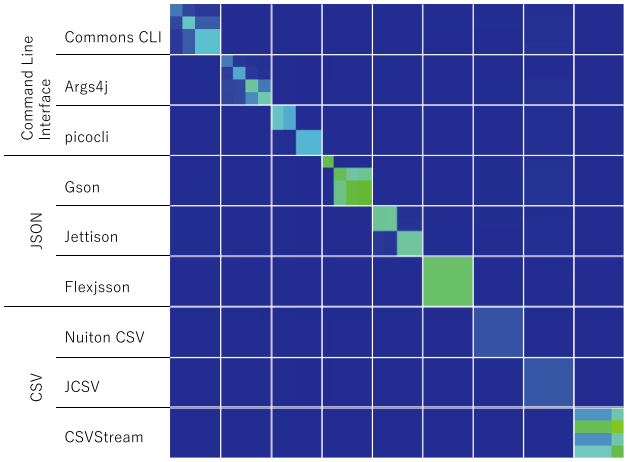


図 9 ProGuard を用いた攻撃耐性評価の結果
Fig. 9 The evaluation result of the tamper resistances against ProGuard.

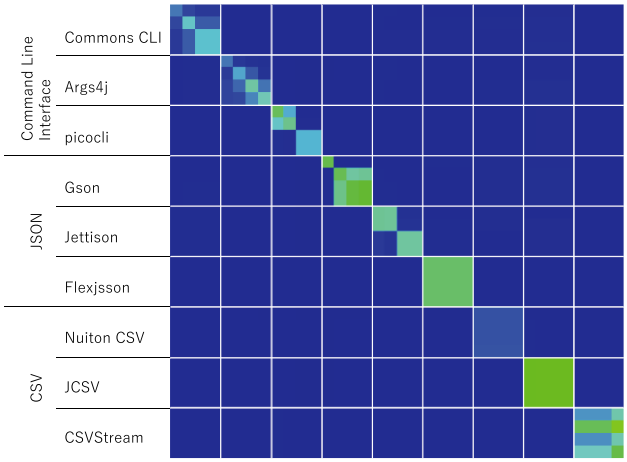


図 10 yGuard を用いた攻撃耐性評価の結果
Fig. 10 The evaluation result of the tamper resistances against yGuard.

ば，JCSV-1.2.0 どうしの類似度が 0.569 から 0.118 と大きく低下した．その原因を調査するため，変換前後から抽出したバースマークおよび，両者のソースコードを調査した．なお，変換後のソースコードを取得するため，CFR [*18] を用いて逆コンパイルを行っている．その結果，ProGuard を適用したことで `java.lang.Enum` の `ordinal` メソッドの呼び出しが削除されたことが分かった．このメソッドは単に定数を返すメソッドであるため，ProGuard を適用することでメソッド呼び出しの代わりに定数利用に処理が置き換わっていた．この `ordinal` メソッドがオリジナルプログラムでは非常に高頻度で呼び出されていたため，このメソッドが削除されたことにより類似度が大きく低下したと考えられる．しかし，図 7 と同じように，異なるアーティファクト間の類似度の最小値は 0.051，平均は 0.013 と非常に小さい．また，JCSV どうしの類似度は低下したものの，$\varepsilon = 0.1$ のときの判定割合は 75%（108/144），$\varepsilon = 0.2$

[*18] http://www.benf.org/other/cfr/

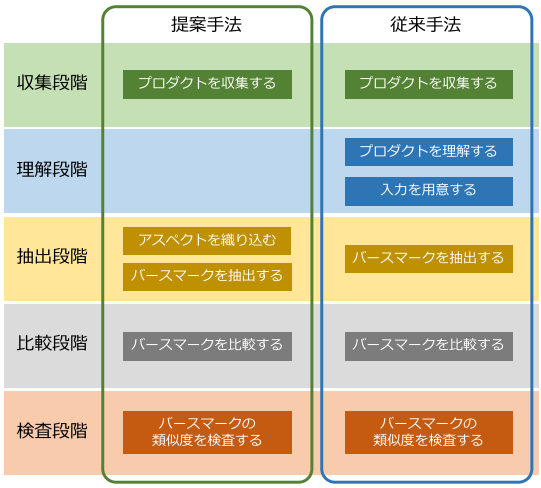


図 11 提案手法と従来手法のフローチャート
Fig. 11 The flowchart of the proposed method and the conventional method.

のとき 37%（53/144），$\varepsilon = 0.3$ のとき 20%（29/144）である．加えて，図 7 と図 9 の結果である類似度の相関係数は 0.8344 と非常に強い相関を示した（$p < 0.0001$）．そのため，$\varepsilon = 0.1$ であれば，提案手法の有用性を大きく損なうものではないと考えられる．

次に，yGuard を適用したときの攻撃耐性評価の結果を確認する．図 10 と図 7 と比較すると，全体的に変化がないことが分かる．図 10 と図 7 の結果である類似度の相関をとると 0.9999 であった（$p < 0.0001$）．そのため，提案手法は yGuard に対して堅牢であるといえる．

5.6 動的バースマークの抽出のコスト評価

5.6.1 動的バースマーク抽出コストの定性的評価

本項では，提案手法による動的バースマークの抽出コストを定性的に評価する．**図 11** は，提案手法と従来手法の違いを示している．提案手法と従来手法の違いは，理解段階と抽出段階のみである．このうち，理解段階にある 2 つの処理，プロダクトを理解する処理と入力を用意する処理は自動化が困難である．一方，提案手法では理解段階に処理を行う必要はない．また，抽出段階で導入されたアスペクトを織り込む処理は自動化可能である．このことから，提案手法では自動化不可能な項目が取り除かれていることが分かる．

動的バースマークは，プログラムの実行時の振舞いから抽出される．したがって，抽出にはプログラムの実行が不可欠であり，対象のプログラムに対して入力を準備する必要がある．従来手法の入力の準備にはプログラムへの理解が必要である．この理解段階は自動化が困難である．定型処理の確立が非常に難しいためである．加えて，入力の準備もプロダクトごとに個別に実施する必要があり，この処理も自動化が困難である．すなわち，この理解段階は自動化が困難であり，非常に大きなコストがかかる．

なお，参考情報として，5.4 節で従来手法を実施したときの状況を以下に記す．5.4 節では，第 1 著者が入力を準備した．入力の準備のため，対象のアーティファクトのホームページ上のドキュメントを参照し，いく度かの試行錯誤のうえ，最終的な入力を定めた．この試行錯誤には，異なるバージョンに対する入力の準備も含まれる．このような入力を 9 つのアーティファクトに対して準備するのに約 12 時間程度要している．なお，入力の準備に要する時間は，入力を準備する利用者のスキル，対象のアーティファクトのドキュメントの質に大きく依存する．ドキュメントに利用例が載せられている場合は，その利用例を踏襲すれば良いものの，記載されていない場合はソースコードの理解も必要となるためである．

一方提案手法では，理解段階の処理は行わない．また，抽出段階がアスペクトの織り込みとバースマークの抽出という 2 つの処理を行うことになる．この点では，抽出段階の処理の数は，従来手法に比べて増加している．しかし提案手法では，アスペクトの織り込みはアスペクトさえ用意されていれば自動化が可能である．加えて，バースマークの抽出もプロダクトに付随する単体テストを実行するだけである．そのため，両処理ともに自動化が容易である．

このように，提案手法はバースマークの抽出までの処理をすべて自動化できるが，従来手法は理解段階の処理を自動化できない．つまり，提案手法はバースマーク抽出処理のコストが，従来手法に比べて削減できているといえる．

5.6.2 アスペクトの準備コストと入力の準備コストの比較

提案手法による動的バースマークの抽出コストは抽出のためのツール開発のコスト v_d と，抽出を実行するコスト v_e に分けられる．抽出のためのツール開発コストのうち，専門知識を要する部分はアスペクトを用意するコストであり，これが v_d のコストの多くを占めると考えられる．そのため，アスペクトを用意するコストを v_d として考える．そして，v_d と x 個のプロダクト $p_i (1 \leq i \leq x)$ の入力を用意するコスト $u(p_i)$ を比較する．アスペクトの開発は自動化が困難であり，かつ一般的に大きなコストを要する．しかし，一般的にプログラム解析ツールや難読化ツールを利用者が作成することはない．そのため，我々の前提として，動的バースマークを抽出するアスペクトはバースマークの開発者（提案者）が作成するものであるとしている．したがって，利用者がアスペクトを作成することはない．一方で，それでも利用者がアスペクトを開発しなければならないケースを考える．この場合，v_d は非常に高いコストであるが，x が非常に巨大な数であれば，$v_d < \sum_{i=1}^{x} u(p_i)$ となると考える．ただし，x が小さなケース，たとえば，盗用の疑いのあるアーティファクトがある程度絞り込めていて，アスペクトが手に入らない場合は，従来手法を適用した方が全体的なコストは下がるであろう．以上のことから，利用者にとってアスペクトの開発に要するコストは不要であ

るといえる．そのため，利用者がバースマークを抽出するには，用意されているアスペクトを利用するだけで良いと考える．

5.6.3 提案手法による抽出コスト

ここでは抽出を実行するコスト v_e について考える．この v_e はツールの準備コスト v_u と実際に実行するコスト v_p に分けられる（$v_e = v_u + v_p$）．v_u はツールのインストールやツールを実行するための調査のような作業にかかるコストのことを指す．一方，v_p はアスペクトを織り込んだプロダクトコードをテストコードを通じて実行するため，単体テストの実行時間となる．したがって，v_p は，v_u に比べると非常に低いものと考えられるため，v_e の多くは v_u で占められるといえる（$v_u \gg v_p$）．この v_u に必要なコストを，提案手法を実装した理想的なバースマーク抽出ツール Rusk を用いて考える．

Rusk は複数の種類の動的バースマークを抽出できるツールとする．そして，特定のプロダクトから特定のバースマークを抽出するとき，Rusk はプロダクトコードの置き場所，テストコードの置き場所，抽出するバースマークの種類の 3 つの入力を必要とする．多くのプロダクトでは，プロダクトコードの置き場所，テストコードの置き場所は分けられており，典型的な置き場所もあるため[*19]，この 2 つの指定に要するコストは低いといえる．また，抽出する動的バースマークの種類の指定は専門的な知識ではあるものの，ツールのドキュメントに利用例を示したり，ヘルプメッセージを表示することで対応可能である．そのため，Rusk を利用するコスト v_u は低いと考えられる．

最後に参考情報として，第 1 著者が 5.4 節の実験を行ったときに要した v_e について述べる．我々は，プロダクトコード，テストコードの場所を指定することで，EXESEQ バースマークを抽出できる簡易版の Rusk を開発しており，これを利用した．このとき，ツールの開発者と利用者が同一であるため，v_u は非常に低いコストとなった．また，v_p は Rusk を実行するだけであるため，プロダクトごとに 1，2 分で終了する．このように，1 つのプロダクトに対し v_e は 2 分程度を要した．

6. まとめ

本稿では，動的バースマークのシナリオの実行時間の短縮を目的とし，事前抽出が可能な動的バースマーク抽出方法を提案した．提案手法では，プロダクトの単体テストコードを利用して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や利用シナリオの有効性を評価した．まず，提案手法によるバースマークどうしでは，同じアーティファクト間では最大 0.999，平均 0.852 と高い類似度であり，異なるアーティファクト間では最大 0.026，平均 0.004 と非常に低い類似度を示した．次に，提案手法と従来手法によるバースマークの比較では，同じアーティファクト間では，最大値 0.629，平均値 0.235 であり，異なるアーティファクト間では，最大値 0.039，平均値 0.013 と低い類似度となった．加えて，ProGuard と yGuard を用いた攻撃に対しても，提案手法と従来手法の比較結果に比べて類似度の大幅な低下は見られなかった．いずれの場合であっても，同じアーティファクト間の類似度が，異なるアーティファクト間と比べると非常に高いため，提案手法によるバースマークは保存性と弁別性を満たしているといえる．また定性的評価から，提案手法は従来手法より抽出コストが低いことを確認した．

本稿では動的バースマーク手法 EXESEQ のみを対象に実験を行った．今後は EXESEQ 以外にも提案手法を適用し，有効性を確認する．また，実験のスケールアップを行い，提案手法と従来手法の比較の際の閾値の目安を調べる．

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

[*19] Maven や Gradle を利用しているアーティファクトの場合，プロダクトコードは `src/main/java`，テストコードは `src/test/java` 以下に置かれることが多い．

参考文献

[1] Tamada, H., Nakamura, M., Monden, A. and Matsumoto, K.: Design and Evaluation of Birthmarks for Detecting Theft of Java Programs, *Proc. IASTED International Conference on Software Engineering* (*IASTED SE 2004*), pp.569–575 (2004).

[2] Tamada, H., Nakamura, M., Monden, A. and Matsumoto, K.: Java Birthmarks—Detecting the Software Theft, *IEICE Trans. Information and System*, Vol.E88-D, No.9, pp.2148–2158 (2005).

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

[4] Tamada, H., Okamoto, K., Nakamura, M., Monden, A. and Matsumoto, K.: Dynamic Software Birthmarks to Detect the Theft of Windows Applications, *Proc. International Symposium on Future Software Technology 2004* (*ISFST 2004*), CD-ROM (2004).

[5] Myles, G. and Collberg, C.: Detecting Software Theft via Whole Program Path Birthmarks, *Proc. Information Security 7th International Conference, ISC 2004*, pp.404–415 (2004).

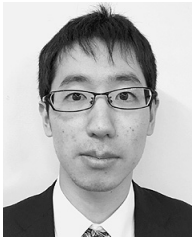
[6] Chan, P.P., Hui, L.C. and Yiu, S.: JSBiRTH: Dynamic JavaScript Birthmark Based on the Run-time Heap, *Proc. 35th Annual Computer Software and Applications Conference* (*COMPSAC*), pp.407–412 (2011).

[7] 横井昂典，玉田春昭：単体テストコードを利用した動的バースマークの抽出，コンピュータセキュリティシンポジウム 2017（CSS2017），3C3-4 (2017).

[8] Yokoi, T. and Tamada, H.: A Beforehand Extraction Method for Dynamic Software Birthmarks using Unit Test Codes, *19th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing* (*SNPD 2018*) (2018).

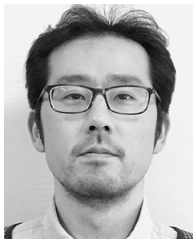
[9] Tsuzaki, T., Yamamoto, T., Tamada, H. and Monden, A.: Scaling Up Software Birthmarks Using Fuzzy

Hashing, *International Journal of Software Innovation* (*IJSI*), Vol.5, pp.89–102 (2017).
[10] 中村 潤，玉田春昭：検索エンジンを用いたソフトウェアバースマークによる検査対象の絞り込み手法，第 24 回ソフトウェア工学の基礎ワークショップ（FOSE2017），pp.99–104 (2017).
[11] Nakamura, J. and Tamada, H.: mituba: Scaling up Software Theft Detection with the Search Engine, *Proc. International Conference on Software Engineering and Information Management* (*ICSIM 2018*), pp.6–10 (2018).
[12] 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).
[13] Tian, Z., Zheng, Q., Liu, T. and Fan, M.: DKISB: Dynamic Key Instruction Sequence Birthmark for Software Plagiarism Detection, *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 (2013).
[14] Schuler, D., Dallmeier, V. and Lindig, C.: A Dynamic Birthmark for Java, *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2007*), pp.274–283 (2007).
[15] Nakamura, J. and Tamada, H.: Fast Comparison of Software Birthmarks for Detecting the Theft with the Search Engine, *Proc. 4th International Conference on Applied Computing & Information Technology* (*ACIT 2016*) (2016).
[16] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-oriented programming, *Proc. European Conference on Object-Oriented Programming* (*ECOOP 1997*), pp.220–242 (1997).
[17] Prasanna, D.R.: *Dependency Injection*, Manning Publications Co., 1st edition (2009).
[18] Kuhn, H.W.: On the origin of the Hungarian method, *History of Mathematical Programming*, pp.77–81 (1991).
[19] Tian, Z., Liu, T., ZHENG, Q., Zhuang, E., Fan, M. and Yang, Z.: Reviving Sequential Program Birthmarking for Multithreaded Software Plagiarism Detection, *IEEE Trans. Software Engineering*, pp.491–511 (2017).



横井 昂典

2017 年京都産業大学コンピュータ理工学部卒業．2019 年同大学大学院先端情報学研究科博士前期課程修了．



玉田 春昭 （正会員）

2006 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．同年同大学産学官連携研究員．2007 年同大情報科学研究科特任助教．2008 年京都産業大学コンピュータ理工学部助教．2013 年同大学同学部准教授．2018 年同大学情報理工学部准教授．ソフトウェアセキュリティ，エンピリカルソフトウェア工学の研究の従事．IEEE, IEICE 各会員．