2005－CSEC－31（6）
2005／12／9

特徴抽出と抽象化による動的バースマークの構成とその検証

林 晃一郎† 楓 基靖† 真野 芳久‡

† 南山大学大学院数理情報研究科 ‡ 南山大学数理情報学部

〒 489-0863 愛知県瀬戸市せいれい町 27

E-mail: †{m04mm006, m05mm012}@msie.nanzan-u.ac.jp, ‡ymano@nanzan-u.ac.jp

概要 プログラムの盗用発見を目的とする動的バースマークを, 実行時情報からの特徴の抽出とその特徴を簡潔に表現する抽象化の 2 つの操作の組合せと考える枠組みが提案されている [4]. この枠組みに従い, それぞれについて条件を満たす候補を検討し組み合わせることによって, 有効かつ利用可能な種々の動的バースマークを構成可能であると期待できる. しかし, 十分な実例による枠組みの有効性の検証はなされていない. 本研究では, この枠組みの有効性の検証を目的として, 抽象化操作の例をいくつか示し, それらに基づく動的バースマークを構成して実験・評価する. 構成された動的バースマークは多くの改変に対して耐性を持つことが確かめられた.

キーワード ソフトウェアバースマーク, ソフトウェアの盗用, ソフトウェアの識別, 動的解析, Java

Feature-Abstraction Framework to Construct Dynamic Birthmarks and Some Experiments

Koichiro Hayashi† Motoyasu Kaede† Yoshihisa Mano‡

† Graduate School of Mathematical Sciences and Information Engineering, Nanzan University

‡ Faculty of Mathematical Sciences and Information Engineering, Nanzan University

27 Seireicho, Seto-shi, Aichi, 489-0863 Japan

E-mail: †{m04mm006, m05mm012}@msie.nanzan-u.ac.jp, ‡ymano@nanzan-u.ac.jp

Abstract A framework has been proposed to construct dynamic software birthmarks for detecting the program thefts [4]. In this framework, birthmarks are constructed by combining two operations, the extraction of some feature from run-time information, and the abstraction of the data with the feature. Since this framework makes us consider each operation individually, we can expect gaining various effective birthmarks. This expectation, however, has not been demonstrated by enough experiments. We construct some birthmarks by proposing some abstraction methods, and demonstrate the effectiveness of the framework. The constructed birthmarks show high resilience for many semantic preserving translations.

Keyword Software Birthmark, Software Theft, Software Identification, Dynamic Analysis, Java

1 はじめに

近年のプログラムの盗用の問題は深刻である. そのため盗用の発見を目的とした技術が要求され, その 1 つにソフトウェアバースマーク (Software Birthmark) [1–4] がある. バースマークとは対象とするプログラムそのものの特徴を表現したもの, またはその表現の一致をもってプログラムの識別や盗用の発見を行う技術を指す. 同様の技術に電子透かし [5] があるが, バースマークは事前に特殊な情報を埋めないため配布済みプログラムにも有効である. しかし秘密情報を埋めない性質上, 盗用とは無縁のプログラム間でバースマークが偶然に一致する可能性を持つ欠点がある. そのため, 一般に 1 つのバースマークだけではなく複数のバースマークの比較をもって盗用を発見する. また, 盗用の事実隠蔽のためにプログラムが改変されても, それから構成可能なバースマークは改変前とほぼ一致することが求められる.

このような要求に対し, プログラムの実行時情報の特徴の表現である動的バースマークを構成するための枠組みが提案されている [4]. その提案では動的バースマークを, 実行時情報からの特徴の抽出とその抽象化という 2 つの操作の組合せから構成する. これにより, (1) 1 つの実行時情報に対し, 抽出する特徴と抽象化方法の 2 つを互いに独立に選択し, 様々に組み合わすことで複数の動的バースマークの候補を構成できる, (2) 実行時情報から改変に強い特徴のみを抽出したり, 抽象化で改変に弱い箇所を捨象するなどの工夫によって, 改変に影響されにくい動的バースマークが構成できる, という 2 つの有効性が期待されている. しかし現在, 抽象化方法の数が乏しい. そのために枠組みに従って構成される動的バースマークは少なく, 上述の 2 つの有効性は実際には検証されていない.

そこで本研究では盗用が近年問題視される Java を対象として実行時情報の特徴の抽象化方法をいくつか提案し, 枠組みによる動的バースマークを構成可能とする. ま

た, 特徴と抽象化方法の選択の組合せから複数の動的バースマークが構成可能であることと, それらの改変への耐性を評価することで, 上述の枠組みの有効性の検証を目標とする. なお, 実行時情報から抽出可能な種々の特徴とその取得方法については, 別の機会に報告する予定である.

2 ソフトウェアバースマークの定義

本節では想定するプログラムの盗用と, バースマークの分類と動的バースマークの定義を述べる.

2.1 ソフトウェアの盗用と攻撃

一般にプログラムの盗用者は盗用の事実を隠蔽するため, 盗用したプログラムに細工をする (以降, 攻撃と呼ぶ). この点に着目し, 本稿では盗用を次に定義する.

定義 2.1 (盗用) プログラム P と P' がある. P' が P の盗用とは P' が P の完全な複製であるか, 複製に攻撃を加えて得られたものである.

なお, 攻撃は最適化や難読化 [5] などのプログラムの外的な振舞いを変更しない機械的な変換を想定する[*1] .

2.2 ソフトウェアバースマークの分類

バースマークは静的バースマーク [1] と動的バースマーク [2–4] に大別できる.

(1) 静的バースマーク ソースコードなどのプログラムの静的な特徴から構成される. アプリケーションの一部分からの構成が可能であるが, 一般に攻撃耐性が低い.

(2) 動的バースマーク プログラムの実行時情報の特徴から構成される. プログラムの表現上の改変に耐性があるが, 実行のために必要なアプリーションの全部分を用いて構成される.

このように両者は互いに短所を補う関係であるが本研究では動的バースマークを扱うとし, 以降, 特に断らない限りバースマークは動的バースマークを指す.

2.3 動的バースマークの定義

本研究では [4] と同様に, 動的バースマークを実行時情報からの特徴の抽出とその抽象化の結果と捉え, 動的バースマークを**定義 2.5** の通りとする.

定義 2.2 (部分系列) 系列 $T = \langle t_1, t_2, \cdots, t_m \rangle, S = \langle s_1, s_2, \cdots, s_n \rangle$ があり, 条件 "$\exists(i_1 < i_2 < \cdots < i_n), \forall j, t_{i_j} = s_j$" を満たすとする. この時 S を系列 T の部分系列と呼ぶ.

定義 2.3 (実行系列) $trace(P, i)$ はプログラム P に入力 i を与えて実行した実行過程の全情報とし, 時系列 $trace(P, i) = \langle t_1, t_2, \cdots, t_m \rangle$ と表す. これをプログラム P の入力 i による実行系列と呼び, この系列中の各要素は何らかの特徴を持ち, 特徴により分類可能とする.

定義 2.4 (抽出系列) S は実行系列 $trace(P, i)$ と等しい系列, または部分系列であると仮定する. S/X は系列 S 中の X の特徴を持つ全ての要素を含みそれ以外の要素を含まない部分系列とし, これを系列 S の特徴 X による抽出系列と呼ぶ.

定義 2.5 (動的バースマーク) 実行系列 $trace(P, i)$ 中で着目する特徴を X, 系列を引数に取り何らかの抽象表現を生成する関数を abs とし, $f(P, i)$ を次の 2 つの手順によって得られる抽象表現とする.

手順 1. 抽出系列 $T = trace(P, i)/X$ を求める.
手順 2. 抽象表現 $f(P, i) = abs(T)$ を求める.

次の 2 つの性質を $f(P, i)$ が満たすならば, $f(P, i)$ をプログラム P の入力 i による動的バースマークと呼ぶ (記号 $\fallingdotseq$ は類似を意味する).

性質 2.1 (保存性) Q が P の盗用 $\Rightarrow f(P, i) \fallingdotseq f(Q, i)$.

性質 2.2 (弁別性) P と Q は互いに独立に実装されている $\Rightarrow f(P, i) \not\fallingdotseq f(Q, i)$.

なお弁別性は, P と Q の仕様が等しい場合でもこれらが独立に実装されたならばバースマークは異なることを意味する. つまり, バースマークは実装者独特のプログラムの特徴をできる限り反映したものが求められる. また, 電子透かしのように特殊な情報を埋めない性質上, バースマークは異なるプログラム間で偶然に一致する可能性を持つ. つまり弁別性は常に満たされるとは限らないため, 一般的に 1 つのバースマークの一致をもって盗用を断定するのは危険であり, 複数のバースマークの一致による盗用の発見が必要とされる.

3 関連研究と動的バースマーク構成の枠組み

本節では従来の研究を述べる. また, 動的バースマーク構成の枠組みを以前の研究と照らしあわせて述べる.

バースマークの初期の研究に Java を対象とした Tamada ら [1] の静的バースマークがある. 彼らはフィールドの初期値, API メソッド呼出しの並び, 継承関係, 使用しているクラス群をバイトコードから抽出して静的バースマークとした.

動的バースマークには Myles ら [2], 岡本ら [3], 古田ら [4] などがある. Myles らは Java のバイトコード中の各基本ブロックの実行パターンを文脈自由文法で表現し, その表現をグラフ化したものをバースマークとした. 岡本らは Windows OS 上で動作するプログラムを対象とし, 実行時の各 Windows API の呼出し順序と頻度をバースマークとした. これらのバースマークは攻撃に高い耐性と弁別性を有することが確認されている.

一方で, 古田らは**定義 2.5** のように動的バースマークを

[*1] 手作業による改変も考えられるが大規模なプログラムの場合, 人がその内容を理解して改変することには限界があるだろう. そのため我々は, ツールなどによる機械的な改変が主流と考える.

実行時情報からの特徴の抽出とその抽象化の結果とし, 従来の動的バースマークをより一般化した. また, 彼らは抽出する特徴と抽象化方法を互いに独立なものとし, これら 2 つを様々に選択して組み合わす枠組みを提案している. これにより特徴の種類の数と抽象化の種類の数との組合せの数のバースマークを検討できる. 枠組みの提案以前は**定義 2.5** のような構成手順は議論されず, バースマークは単にプログラムの変更されにくい特徴を列挙した結果とされた傾向にある. そのため, 抽象化はあまり意識して行われなかったり, 抽出する特徴と抽象化方法は互いに独立なものとして検討はされてこなかった. Myles ら [2] のように実行パターンをグラフ化するような抽象化が行われたものもあるが, 特徴と抽象化方法を様々に組み合わせてバースマークを構成することは意図していない. このように枠組みによる動的バースマークの構成では, 従来のバースマークとは異なる新たな方向性を与えている.

4 特徴の抽象化方法の提案

3 節で動的バースマーク構成の枠組みについて述べたが, [4] では枠組みに適用可能な抽象化方法を 1 つしか提案していない. また, その他の従来の研究で用いられている抽象化方法は, 様々な特徴との組合せを想定したものではない. そのため現在, 枠組みに従って構成可能なバースマークはほとんどなく, 枠組みの有効性の検証もなされていない. 本節ではこの問題の解決を目的として, 枠組みに適用可能な抽象化方法を提案する. まず, 4.1 節では [4] で提案された抽象化方法を紹介する. 4.2 節では実行系列中での処理の間隔を利用した抽象化, 4.3 節では各処理のばらつきを利用した抽象化を新たに提案する.

4.1 繰返しパターンによる動的バースマーク

プログラムには繰返し文や再帰などから実行時に同じ処理が連続する箇所がある. 繰返し内の細部の処理は攻撃を受ける可能性がある[*2] が, 実行時の情報の繰返し回数や実行順序の機械的な変更は一般に困難である. [4] はこれらの点から実行系列を繰返しパターンで抽象化する方法を提案し, 動的バースマークを構成している.

具体例で抽象化の方法を述べる. $trace(P,i)/X = \langle a,b,c,a,b,c,d,e,d,e,a,b,c,a,b,c,d,e,d,e\rangle$ という抽出系列を得たとする. メタ記号 (,), ^ を (,) はグループ化, ^n は直前要素の n 回の繰返しとすると, 上記の系列は $((a,b,c)\hat{}2(d,e)\hat{}2)\hat{}2$ と表現できる. この表現に対し, 繰返し内の処理 a,b,c,d,e の除去を行い, $((\)\hat{}2(\)\hat{}2))\hat{}2$ という抽象表現をバースマークとする.

抽出系列 $trace(P,i)/X$ の特徴 X は例えばメソッド呼出しやフィールド変数値の変更などを選択でき, それぞれの特徴から別のバースマークが構成できる. [4] ではこれら 2 つの特徴によるバースマークを実際にそれぞれ構成し, 評価している. その結果, 多くの難読化に対して難読化を施したプログラムからも難読化前と同様のバースマークを得ている. これは抽象化で攻撃に弱い繰返し内の処理を捨象したため, バースマークが難読化後に変化しにくいと考えられる. また, 弁別性を満たすことも確認されている. メソッド (API メソッドは除く) やフィールド変数に対しては様々な難読化が提案されており, 従来ではこのような攻撃に耐性が低い特徴はバースマークの構成に利用されなかった. しかし, ここで述べた抽象化を用いることでこれらの特徴も構成の対象にできる.

4.2 出現間隔による動的バースマーク

必要不可欠な処理であっても, 実装者によりそれらの処理を実行させる位置やその記述方法は異なる. そのため, 実装者が異なれば同じ仕様のプログラムでも各処理の実行時点は異なると考える. また, 改変による処理の実行時点の大幅な変更は仕様を破壊しかねないため機械的には困難である. そこで, 実行系列中のある処理 (特徴) に着目し, その処理の各実行の間隔による抽象化を用いた動的バースマークを**定義 4.4** で提案する.

定義 4.1 (不変な名前) プログラム中の識別子で攻撃によって容易に変更されにくいものを不変な名前と呼ぶ.

定義 4.2 (要素間距離) 系列 $T = \langle t_1, t_2, \cdots, t_m\rangle$ がある. 系列 T の要素 t_i, t_j $(1 \le i, j \le m)$ の要素間距離を $dist_T(t_i, t_j) = |j - i|$ と定める.

定義 4.3 (出現間隔列) 実行系列 $trace(P,i)$ の部分系列を $T = \langle t_1, t_2, \cdots, t_m\rangle$, A をある不変な名前 a を持つという特徴とし, $T/A = \langle t_{j_1}, t_{j_2}, \cdots, t_{j_n}\rangle$ $(1 \le j_1 < j_2 < \cdots < j_n \le m)$ とする. 系列 T に対する不変な名前 a の出現間隔列 $interval_T(a)$ を次で定める

$$interval_T(a) = \langle dist_T(t_{j_1}, t_{j_2}), \cdots, dist_T(t_{j_{n-1}}, t_{j_n})\rangle$$

定義 4.4 (出現間隔バースマーク) k 個の不変な名前 $a_1, a_2, \cdots, a_k$ が系列 $T = trace(P,i)/X$ の要素のいくつかに特徴として含まれるとする. $I_j = interval_T(a_j)$ $(1 \le j \le k)$ とおき, 順序組 $(I_1, I_2, \cdots, I_k)$ を系列 T の不変な名前 $a_1, a_2, \cdots, a_k$ による出現間隔バースマークと呼ぶ.

このように**定義 4.4** では, 実行系列中での不変な名前の位置を間隔で表現する抽象化を提案している.

特徴 X の例

定義 4.4 の提案では, 特徴 X は不変な名前を含んだ特徴であることを要求している. Java プログラムでの不変な名前は API メソッド名や API クラス名に相当する. なぜなら API 関連の改変は, そのライブラリの解析やライブラリ自体の改変が生じ, 非常に困難なためである[*3] .

[*2] 変数名やメソッド名の変更, 変数やフィールドの分割や統合, メソッドのインライン化など [5].

[*3] 一方で, ユーザ定義メソッドやユーザ定義クラスは名前の変更が容易であり, 他にも多数の難読化手法が提案されている [5].

以上を踏まえると, 特徴 X は例えばメソッド呼出し, クラスの使用などがある. また, API クラスを雛型とするインスタンスも, その特徴に不変な名前 (API クラス名) を持つといえる. そのため, インスタンスへの参照も X の選択の対象となる. また, ユーザ定義のクラスもそのスーパークラスに API クラスを持っていれば, 不変な名前を特徴として持つと扱える. このように間接的に API と関連するものも不変な名前の特徴を持つと捉えれば, 抽出特徴の選択の範囲を広げられる.

バースマークとしての妥当性

前述したように実装者により処理を実行する時点は異なる. そのため, 出現間隔列にその違いが表れるであろう. また, 実行時点の大幅な変更は仕様の破壊につながるため困難である. 実行系列の一部だけを攻撃しても, 一部の要素間距離が変化するだけで各出現間隔列の全体は大きく変化しないとも考えられる. さらに不変な名前を特徴として持つプログラムの構成要素や処理は, 改変が困難である. 以上から, 提案したバースマークは保存性と弁別性を満たすことが期待できる.

バースマークの比較方法

抽出系列 $T = trace(P,i)/X$, $T' = trace(P',i)/X$, 不変な名前 $a_1, \cdots, a_k$ があるとし, $I_j = interval_T(a_j) = \langle x_1, \cdots, x_{m_j} \rangle, I'_j = interval_{T'}(a_j) = \langle y_1, \cdots, y_{n_j} \rangle$ $(1 \leq j \leq k)$ とおく. そして 2 系列の最長共通部分列 [6] の長さを $L_{I_j,I'_j}(m_j, n_j)$ (以降 I_j と I'_j は省略) として一致率 R_j を次で定義する.

$$R_j = \frac{2 \cdot L(m_j, n_j)}{m_j + n_j}$$

なお, $L(m_j, n_j)$ は次の規則で与えられる.

$$L(m_j, n_j) = \begin{cases} L(m_j - 1, n_j - 1) + 1 & (x_{m_j} = y_{n_j}) \\ \max\{L(m_j, n_j - 1), L(m_j - 1, n_j)\} & (x_{m_j} \neq y_{n_j}) \end{cases}$$

こうして求めた $R_1, \cdots R_k$ の多くが 1 に近い数値の場合, プログラム P と P' は同一のものである可能性があるとする. なお, 求められた一致率のうちどれだけが一般に高い数値を示すべきかは調査中である.

4.3 近傍検索による動的バースマーク

お互いに近い時点で実行される処理同士は関連がある場合が多く, 一連の処理のまとまりの機械的な変更は仕様を破壊する危険がある. つまり, 各処理の実行時点をお互いに大幅に引き離すことは困難であろう. そこで, お互いに近い時点に現れる処理 (特徴) 同士の関係をグラフとして抽象化する動的バースマークを**定義 4.6** で提案する.

定義 4.5 (近傍集合) 実行系列 $trace(P,i)$ の部分系列を $T = \langle t_1, t_2, \cdots, t_m \rangle$, A をある不変な名前 a を持つ特徴とし, $T/A = \langle t_{j_1}, t_{j_2}, \cdots, t_{j_n} \rangle$ とおく. $name(e)$ で実行系列中の要素 e が特徴として持つ不変な名前を表し, e が不変な名前を特徴として持たない場合は空とする. d を定数とし, 系列 T に対する不変な名前 a の近傍集合 $N^d_T(a)$ を次で定める.

$$N^d_T(a) = \{name(t_p) | 1 \leq \exists p \leq m, 1 \leq \exists q \leq n, dist_T(t_p, t_{j_q}) \leq d\}$$

これは次の性質を満たす.

性質 4.1 $a \in N^d_T(b) \Leftrightarrow b \in N^d_T(a)$

定義 4.6 (近傍検索バースマーク) k 個の不変な名前を $a_1, a_2, \cdots, a_k$, 抽出系列 $T = trace(P,i)/X$ とする. d を定数として $V_j = N^d_T(a_j)$ $(1 \leq j \leq k)$ とおき, 順序組 $(V_1, V_2, \cdots, V_k)$ を系列 T の不変な名前 $a_1, a_2, \cdots a_k$ による距離 d の近傍検索バースマークと定める.

このように**定義 4.6** では, 実行系列中の不変な名前の近傍に出現にする他の不変な名前の集合を用いた抽象化を提案している.

特徴 X の例

定義 4.6 は不変な名前を持つ特徴を用いる. そのため 4.2 節と同様に, 特徴 X は不変な名前を含む特徴を選択の対象とする.

バースマークとしての妥当性の考察

同じ仕様のプログラムでも実装者により処理の実行位置や記述方法は異なる. そのため実行系列中の不変な名前のばらつきは実装者により異なり, 近傍集合にその違いが反映されると考えられる. また, プログラム中の一連の処理のまとまりの機械的な変更は困難である. そのため, 攻撃で処理の実行順序が交換されるなどしても, 各処理の実行時点が互いに大幅に離れることは少ないと想定できる. このような多少の順序交換等の攻撃に対し, 近傍集合は実行系列中で不変な名前が出現する順序に依存しないため影響を受けにくいといえる.

バースマークの比較方法

抽出系列 $T = trace(P,i)/X$, $T' = trace(P',i)/X$, k 個の不変な名前 $a_1, a_2, \cdots, a_k$ があるとし, d を定数として, 一致率 R^d を次で定義する.

$$R^d = \frac{2\sum_{j=1}^{k} \left| N^d_T(a_j) \cap N^d_{T'}(a_j) \right|}{\sum_{j=1}^{k} \left(\left| N^d_T(a_j) \right| + \left| N^d_{T'}(a_j) \right| \right)}$$

こうして求めた R^d が 1 に近い数値の場合, プログラム P と P' は同一のものである可能性があるとする.

5 ソフトウェアバースマークの評価について

本節では 4 節で提案したバースマークを実際に構成し, 評価を行う. 5.1 節ではバースマークの評価方法を, 5.2 節では各バースマークの攻撃耐性の評価結果を述べる.

5.1 評価方法

バースマークが満たすべき性質として, 保存性 (**性質 2.1**) と弁別性 (**性質 2.2**) がある. 評価ではこれらの確認が必要である.

保存性の評価方法

保存性はバースマークの攻撃耐性を評価尺度とする．バースマークは攻撃耐性があるほど有効である．一方, 難読化ツールとして SandMark [7] があり, これはプログラムに種々の難読化を施せる．本研究では SandMark を用い, プログラムに難読化を施した後のバースマークの変化の度合を調べる．

弁別性の評価方法

弁別性の評価は, 独立に実装され, かつ仕様や構造が類似するプログラムを複数用意して, これらから構成されるバースマーク同士を比較して行う．この場合, バースマークの一致が小さいほど有効である．

5.2 攻撃耐性の評価結果

定義 4.4 と**定義 4.6** のバースマークをそれぞれ構成し, 実際に攻撃耐性を評価した．なお, 評価に用いた Java プログラムは両者とも jdepend-2.9.1.jar [8] である．

5.2.1 出現間隔バースマークの攻撃耐性

構成するバースマークは特徴 X を全ての API メソッド呼出しとし, 抽出系列 $trace(P,i)/X$ 中の要素のいくつかが特徴として持つある 15 個の API メソッド名の出現間隔列の順序組とする．なお, 15 個の API メソッド名は抽出系列 $trace(P,i)/X$ 中の特徴として 100 回以上出現するものを用いた．そして, 難読化前と難読化後のもので 15 個の各出現間隔列同士の一致率を計測した．その結果を**表 1** に示す．なお, 表の難読化方法は SandMark が提供する難読化の種類, 一致率の個数は難読化前と難読化後の各出現間隔列を比較して一致率 (少数点第 3 位以下は四捨五入したもの) が 0.90〜0.94, 0.95〜0.99, 1.00 の 3 つの範囲に収まる個数を示す．表のほとんどの難読化に高い耐性を示している．これは実行系列から抽出する特徴 X をここでは API メソッド呼出しを選択したため, 攻撃影響を受けにくいと考えられる．なぜなら 4.2 節で述べたように, API メソッドはその名前の変更やインライン化などの様々な攻撃に耐性を持つからである．また, 高い一致率を示した難読化方法の多くは, 各ユーザ定義メソッドの表現の変更やそれらのインライン化, または実行されないダミーコード挿入などである．つまり, これらの難読化は実行系列中の API メソッド呼出しの順序や頻度に影響を与えないため, 難読化後も出現間隔列がほとんど変化しないと考えられる．

しかし, API メソッド呼出しを余分に, しかも大量に追加する攻撃には耐性が低い．これは Promote Primitive Types と Split Classes の難読化結果として表れている．Promote Primitive Types は `int` 型などのプリミティブ型を `java.lang.Integer`, いわゆるラッパークラスの型に変換する難読化である．ラッパークラスで覆われた数値を取り出すには API メソッド `java.lang.Integer.intValue()` などを呼び出す必要があり, このようなメソッド呼出しがプログラム全体に追加されたため出現間隔列の破壊が生じたと考えられる．Split Classes はクラスを複数のクラスに分割する難読化で, 実行時のクラスロード処理が難読化前と比べ増える．クラスロード処理では API メソッド `java.lang.ClassLoader.loadClassInternal(String)` を呼び出すため, これが余分にプログラム全体に追加されて一致率が低くなったと考えられる．以上の通りいくつかの攻撃には耐性が低い．しかし現実には, 全ての攻撃に耐性があるバースマークの構成は困難であるため, 他のバースマークも用いて弱点を補うことが必要である．

表 1　出現間隔列の攻撃による変化

難読化方法	一致率の個数		
	0.90〜0.94	0.95〜0.99	1.00
BLOAT	0	0	15
Bludgeon Signatures	0	0	15
Buggy Code	0	0	15
Dynamic Inliner	0	1	14
Inliner	0	1	14
Irreducibility	0	0	15
Objectify	0	0	15
Opaque Branch Insertion	0	0	15
Overload Names	0	0	15
Promote Primitive Types	0	0	0
Reorder Parameters	0	0	15
Split Classes	1	4	1
Static Method Bodies	0	0	15

表 2　近傍検索バースマークの攻撃による変化

難読化方法	一致率			
	R^5	R^{10}	R^{15}	R^{20}
BLOAT	1.00	1.00	1.00	1.00
Bludgeon Signatures	1.00	1.00	1.00	1.00
Buggy Code	1.00	1.00	1.00	1.00
Dynamic Inliner	0.90	0.92	0.93	0.93
Inliner	1.00	1.00	1.00	1.00
Irreducibility	1.00	1.00	1.00	1.00
Objectify	1.00	1.00	1.00	1.00
Opaque Branch Insertion	1.00	1.00	1.00	1.00
Overload Names	1.00	1.00	1.00	1.00
Promote Primitive Types	0.84	0.84	0.87	0.87
Reorder Parameters	1.00	1.00	1.00	1.00
Split Classes	0.99	0.98	0.98	0.99
Static Method Bodies	0.85	0.74	0.75	0.76

5.2.2 近傍検索バースマークの攻撃耐性

次に構成するバースマークは特徴 X を全てのメソッド呼出し (ユーザ定義メソッドと API メソッドの両方を含む) とし, 抽出系列 $trace(P,i)/X$ の要素に特徴として含まれる全ての API メソッド名 $a_1,\cdots,a_k$ の各近傍集合の順序組とする．そして, 難読化後のプログラムでも同様に, $a_1,\cdots,a_k$ の各近傍集合の順序組を求め, 難読化前のものと比較して一致率を計測した．その結果を**表 2** に示す．表の難読化方法は SandMark が提供する難読化の種類である．一致率は R^5, R^{10}, R^{15}, R^{20} の 4 つを示す．表のほとんどの難読化に対して高い耐性を示してい

る．5.2.1 節で選択した特徴 X は API メソッド呼出しのみであったが, ここでは X をユーザ定義メソッド呼出しも含めて選択している．そのため, ユーザ定義メソッドのインライン化 (Dynamic Inliner, Inliner [*4]) によって抽出系列 $trace(P,i)/X$ は変化すると予想されるが, それでも 4 つの各一致率は 0.9 以上を示している．また, 5.2.1 節ではほとんど一致が起きなかった Promote Primitive Types と Split Classes に対しても, 各一致率は 0.8 以上を示している．これらの結果は, 抽出系列 $trace(P,i)/X$ を抽象化で攻撃に影響されにくい表現へ変換したため得られたと考えられる．近傍集合はある範囲内に散らばる不変な名前の集合であり, これによって多少の処理の実行時点の変更に影響を受けない．

しかし, Static Method Bodies については若干, 一致率が低くなっている．これはプログラム中のコードの断片のいくつかをモジュール化する．つまり, メソッドを用意してその中にコードの断片をまとめてしまう．これによりプログラム中のメソッドの総数が大幅に増し, 同時に実行時に呼び出されるメソッドの総数も増す．そのため, 近傍集合に多少の違いが表れたと考えられる．

6 まとめ

本稿のまとめと今後の課題を述べる．本稿ではバースマークを実行時情報からの特徴の抽出とその抽象化という 2 つの操作の選択的な組合せと捉える枠組みを取り上げた．そしてこの枠組みの有効性の検証を目的として, 抽象化操作の例として出現間隔バースマークと近傍検索バースマークを新たに提案し, それらに基づくバースマークを実際に構成して攻撃耐性を評価した．評価したバースマークは多くの改変に対して耐性を示した．

今後の課題について述べる．抽象化方法を 4 節で新たに提案したが, 弁別性の評価はまだ行っていない．また, 特徴と抽象化方法を様々に組み合わすことで実際に複数のバースマークが構成可能であることを確認していない．今後はこれらの評価を進める必要がある．また, 本稿で提案した抽象化方法について, 評価結果を基に性質を分析し, より有効なバースマークを構成できるように補正する必要があるだろう．

参考文献

[1] Haruaki Tamada, Masahide Nakamura, Akito Monden, “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, 2004.

[2] Ginger Myles, Christian S. Collberg, “Detecting Software Theft via Whole Program Path Birthmarks”, Information Security Conference, pp.404-415, 2004.

[3] 岡本 圭司, 玉田 春昭, 中村 匡秀, 門田 暁人, 松本 健一, “ソフトウェア実行時の API 呼び出し履歴に基づく動的バースマークの実験的評価”, プログラミング・シンポジウム, pp.41-50, 2005.

[4] 古田 壮宏, 真野 芳久, “実行系列の抽象表現を利用した動的バースマーク”, 電子情報通信学会論文誌 D-I, Vol.J88-D1, No.10, pp.1595-1598, 2005.

[5] Christian S. Collberg, Clark Thomborson, “Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection”, IEEE Transactions on Software Engineering, Vol.28, No.8, pp.735-746, 2002.

[6] D. S. Hirschberg, “Linear Space Algorithm for Computing Maximal Common Subsequences”, Communications of the ACM, Vol.18, No.6, pp.341-343, 1975.

[7] “SandMark: A Tool for the Study of Software Protection”, http://sandmark.cs.arizona.edu/.

[8] “JDepend”, http://www.clarkware.com/software/JDepend.html.

[*4] Dynamic Inliner はインスタンスメソッドを, Inliner はクラスメソッドをインライン化する．