Javaプログラムの部分盗用に対する動的バースマーク

M2005MM012 楓 基靖
指導教員　青山 幹雄

1　はじめに

1.1　背景

近年，ソフトウェアを不正に復元・解析し，あたかも自分が作成したソフトウェアのように公開するケースが後を絶たず深刻な問題となっている．特に，プラットフォーム独立や再利用性などの特性を持つ Java 言語は逆コンパイラなどによる復元・解析が容易であり，改変や盗用される危険性が高い．

1.2　盗用対策技術

これまでに盗用を抑止・発見する技術として，難読化，電子透かし，バースマークが提案されている．難読化は逆コンパイラなどによるソースの復元・解析にかかるコストを増やし，事前に盗用を抑止する技術である．電子透かしはデータに対して著作権情報 (透かし情報) を埋め込み，盗用後に透かし情報を用いて真正証明を行う技術である．バースマークは，プログラムが持つ特徴の一致をもって盗用発見の可能性を示す技術である [1]，[2]，[3]，[4]，[5]．バースマークは難読化や電子透かしとは異なり，配布前に予め処理を施す必要がないため既に配布されているものに対しても有効である．また，実際にプログラムを実行させることで得られる実行系列を基に構成する動的バースマークは，盗用者によるプログラム構造などの改変による攻撃に耐性を持つとされている．

1.3　目的

動的バースマーク (以下バースマークと略記) は配布済みのプログラムに対しても有効であり，また攻撃に耐性を持つことから盗用発見に有効であると考えられる．しかし，従来のバースマークは部分盗用に対する議論がなされていない．本研究では，盗用され易い Java 言語プログラムを対象に，従来のバースマーク [4]，[3] を応用し部分盗用の発見を行う手法を提案する．

2　部分盗用の定義

本研究で想定する部分盗用の定義を示す．

プログラム P と Q がある．P，Q はそれぞれ，複数の要素から構成され，$P = \{P_1, P_2, ..., P_i, ..., P_n\}$，$Q = \{Q_1, Q_2, ..., Q_j, ..., Q_m\}$ と表現できる．ある i と j について，次の 2 つの条件のうち少なくとも 1 つを満たすならば，Q を P の部分盗用と言い，この事実を $Q_j <\equiv_{theft} P_i$，この事実が成り立たないことを $Q_j <\not\equiv_{theft} P_i$ と表す．

- Q_j は P_i の複製．
- Q_j は P_i の複製に変換を施したもの．

$P_i <\equiv_{theft} Q_j$ または $Q_j <\equiv_{theft} P_i$ の事実が成り立つ場合，P と Q は部分盗用の関係にあると言う．

バースマークは以下の保存性と弁別性を満たすことが望ましい．盗用関係にある P_i と Q_j について，$f(P_i)$，$f(Q_j)$ を仮にそれぞれのバースマークとした場合，保存性と弁別性は以下で定義される．

保存性　$P_i <\equiv_{theft} Q_j$ または $Q_j <\equiv_{theft} P_i$
$\Rightarrow f(P_i) \fallingdotseq f(Q_j)$.

弁別性　$P_i <\not\equiv_{theft} Q_j$ かつ $Q_j <\not\equiv_{theft} P_i$
$\Rightarrow f(P_i) \not\fallingdotseq f(Q_j)$.

また想定する部分盗用の特徴として，盗用単位，実装後の仕様，攻撃の度合いをそれぞれ以下に述べる．

2.1　盗用の単位

一般的に Java プログラムはパッケージやクラスなどの集合として構成される．パッケージやクラスは機能的なまとまりであり，それぞれの単位毎で独立に機能を実現することで再利用が可能となっている．そのため，これらの有用な機能を提供する部分が盗用の被害を受けやすいと考えられる．

よって，本研究で想定する盗用の被害を受ける単位をパッケージ単位，およびクラス単位と定める．

2.2　実装後の仕様

従来の盗用単位はプログラム全体であったため，オリジナルと盗用後のプログラム仕様は同じであった．しかし，部分盗用は再利用可能な単位で行われる．そのためオリジナルからクラスなどを盗用しプログラム本体だけを盗用者が実装した場合，プログラム仕様が異なってしまう．仕様が異なると，盗用部分の利用方法 (パラメータや実行時点) が変わってしまい，得られる実行系列が異なることでバースマークによる盗用発見が困難となる．また同一仕様への実装に利用されても，本体部分は盗用者によって実装されたものであるため，盗用部分がオリジナルと全く同じ方法で利用されるとは限らない．

しかし，同一仕様の方が異仕様に実装された場合より，実行系列は似てくると期待できる．よって本研究では，部分盗用によって実装されるプログラムの仕様が同一，および異なる 2 つの状況をそれぞれ想定する．

2.3　攻撃の度合い

盗用後にその事実を隠蔽するためにプログラムに対して攻撃を行うことが考えられる．攻撃とは，一般的にプログラムの制御構造などを複雑にし解析を困難にすることであり，難読化を悪用することで容易に行うことができる．しかし，手作業で攻撃を行うには膨大なコストが必要であると考えられるため現実的と言えない．そこで本研究で想定する攻撃は，難読化ツール[*1]を用いて行わ

[*1] 既存の難読化アルゴリズムを幅広く搭載していることから SandMark を用いた．

れるものと考える．

ここで，ある特徴に着目したバースマークは，多種多様に存在する攻撃全てに対して耐性を持つことは期待できない．そのため複数のバースマークによって盗用の発見を行うことが必要である．よって本研究ではどの難読化アルゴリズムによる攻撃が盗用隠蔽に効果があるかは議論しない．

3 バースマークの応用方法の提案

3.1 バースマーク構成の枠組みの拡張

バースマークはこれまでに様々なものが提案されているが，本研究では部分盗用への応用に [4]，[3] で提案されたバースマーク構成の枠組みを用いた構成方法を拡張して利用する．この枠組みは，プログラムの実行系列 $trace(P, I)$ から着目する特徴 X と抽象化方法 abs の組み合わせによって系統的に種々のバースマークを構成できるとしている．しかし，ここで着目する特徴 X とは実行系列中のあらゆる場所に存在する可能性があり，盗用部分だけに着目した部分的な系列を得ることができない．

本研究では実行系列から着目した情報を抜き出す抽出 $trace(P, I)/X$ と併せ，新たにプログラムのある部分 P_i の実行系列を抽出するように枠組みを拡張する．また，抽出系列から P_i に関連する情報だけを抜き出した系列を部分系列と呼び以下で表現するものとする．

$$E = trace(P, I)/(X * P_i)$$

ある部分 P_i の部分系列は，実行中の様々な位置で複数回出現して得られる．P_i の実行側の実装者が異なる場合やプログラム仕様が異なると，必然的に実行回数や順序，さらに，引数が異なれば部分系列そのものも変化する可能性が高い．

3.2 部分系列の抽出方法

盗用単位であるパッケージやクラスは，Java プログラムの構造上それ自体は単独で実行を行わない．実行はメソッド単位で行われるため，盗用部分で定義されたメソッドに着目して，そのメソッドの開始から終了までの実行系列を抜き出せばよいと考えられる．しかし，盗用部分の実行方法が変わるとメソッドの実行順序も変わるため，実行順序通りに抽出するとかえってオリジナルと盗用の系列に違いが生じる可能性がある．そこで，本研究では盗用部分で定義されているメソッド毎に実行系列を抽出しメソッド単位で部分系列を得る．メソッドの部分系列に抽象化を行い，得られた特徴の集合をクラス・パッケージのバースマークとし盗用の発見を行う．

3.3 抽象化の検討

一般的に，同一メソッドであってもその実行方法が異なれば，その都度異なる実行系列が得られると考えられる．しかし，実際にはメソッドの定義内容自体は変化していないため，異なる実行系列同士でも何らかの共通の特徴を持つと期待できる．この共通の特徴を得るために，本研究では繰り返しパターン [4]，出現間隔 [3]，近傍集合 [3] による抽象化を行う．しかし従来までの抽象化方法では，複数回実行されたメソッド実行系列から構成される部分系列の抽象化に適していないため，これらの応用方法について検討を行う必要がある．以下で繰り返しパターンを例にその応用方法を述べる．

繰り返しパターンによる抽象化は実行系列中における繰り返し部分を正規表現に似た入れ子表現に簡略化する方法である．一般的にメソッドは for 文や if 文などの繰り返しや条件分岐によってその機能が実現されるため，繰り返しパターンによってその特徴が得られると期待できる．複数回分のメソッドの実行系列から共通する特徴を得るために，まず**図 1(a)** で示すようにメソッド 1 回分の実行系列毎に繰り返しパターンで抽象化を行う．

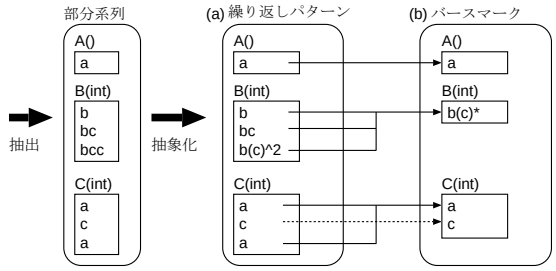

図 1　繰り返しパターンによる抽象化の応用例

メソッド内部に存在する繰り返し回数のみが異なる抽象表現は，繰り返し回数を除くと同一の抽象表現として扱える．異なるパターン同士は，実行方法 (特にパラメータ) の違いによる条件分岐の違いによるものであると考えられる．よって，それぞれがそのメソッドの持つ実行パターンであり一つの特徴であると考えられるため省略できない．最終的に，図 1(b) のように冗長な系列を省いた実行パターンの集合を抽象表現として得る．

例えば $B(int)$ を見ると，それぞれの実行系列は b, bc, bcc であり，これらの共通の特徴はまず b が，その後ろに c が複数回出現する，という情報となり $b(c)^*$ として一つのパターンにまとめることができる．また，$C(int)$ を見る．実行系列は a, c, a であり，複数回出現している a, a はまとめて a とする．a と c はそれぞれ異なるパターンであるので $C(int)$ は 2 種類のパターンが存在することになり，最終的に a, c が得られる．

部分盗用においてはメソッドの実行方法が変化し易く，それによって実行順序や実行回数が変化する．よって，実行パターンに同一の繰り返しが複数存在しても，その出現回数は余り意味がない．出現回数を除いても，得られる実行パターンはメソッドの制御構造を反映したものであり，メソッドの部分系列の抽象表現として妥当である．

4 実験方法と実験結果

部分盗用において，最も実行系列が変化しやすい状況は異仕様へ実装された場合である．よって，実験は同一

仕様に実装された場合と異仕様に実装された場合に分けて行う．それぞれの状況において攻撃を行った後に，盗用クラスおよびパッケージがどの程度の一致を示すかを求める．着目する特徴 X は標準 API メソッド呼び出し情報を扱う．また抽象表現同士の比較には最長共通部分列を用いて一致率を求める．

4.1 仕様が同一の場合

仕様が同じで実装に利用するパッケージやクラスも同じならば，実装者が異なっていても，盗用部分の実行方法は似ると期待できる．保存性と弁別性の実験に分けて，それぞれ，以下で述べる盗用状況を擬似的に再現し，提案手法による実験を行った．

4.1.1 保存性の確認実験

盗用状況を再現するために，ここでは Jakarta や GNU が提供するライブラリ[*2]を用いる．具体的には，Jakarta によるライブラリ L_1 [*3]を用いて実装者 A に仕様 1 [*4]のプログラム $P_{(L_1,A)}$ を実装させる．次に，実装者 B に L_1 を用いて仕様 1 のプログラム $P_{(L_1,B)}$ を実装させる．$P_{(L_1,A)}$ がオリジナル，$P_{(L_1,B)}$ が盗用によって実装されたプログラムである．またライブラリ L_1 が盗用部分である．提案する方法によって得られたパッケージの抽象表現同士の比較結果から一部抜粋したものを**表 1** に示す．また以降の表中の抽象化 1 は繰り返しパターン，抽象化 2 は出現間隔，抽象化 3 は近傍集合による抽象化を示す．

表 1　同一仕様における抽象表現の保存性確認

	抽象化 1	抽象化 2	抽象化 3
改変なし	0.95	0.63	0.96
ArraySplitter	0.21	0.26	0.15
BranchInverter	0.95	0.63	0.96
ClassSplitter	0.71	0.47	0.72
ConstantPoolReorderer	0.95	0.63	0.96
PromotePrimitiveTypes	0.13	0.03	0.05
ReorderParameters	0.95	0.63	0.96
SimpleOpaquePredicates	0.95	0.63	0.96

繰り返しパターンと近傍集合は高い一致率を示している．また，難読化による攻撃のうち 28 種類に対しては，攻撃なしと同じ一致率が得られ耐性を持つことが示された．しかし，標準 API 呼出しが増える一部の難読化手法では一致率が著しく低下する．これは，盗用部分のメソッド内部に新たな実行が増えることでパターンそのものが変化した結果であると考えられる．

4.1.2 弁別性の確認

弁別性を確認するために，GNU のライブラリ L_2 [*5]を用いて実装者 B に仕様 1 のプログラム $P_{(L_2,B)}$ を実装させる．$P_{(L_1,A)}$，$P_{(L_2,B)}$ は完全に独立で実装された非盗用関係である．この盗用状況における $P_{(L_1,A)}$ と $P_{(L_2,B)}$ の比較結果から一部抜粋したものを**表 2** に示す．

表 2　同一仕様における抽象表現の弁別性確認

	抽象化 1	抽象化 2	抽象化 3
改変なし	0.35	0.03	0.10
ArraySplitter	0.19	0.05	0.05
BranchInverter	0.34	0.03	0.10
ClassSplitter	0.21	0.03	0.06
ConstantPoolReorderer	0.34	0.03	0.10
PromotePrimitiveTypes	0.11	0.01	0.01
ReorderParameters	0.34	0.03	0.10
SimpleOpaquePredicates	0.34	0.03	0.10

表 1 の盗用関係の結果と比べて，全体的に一致率が低い結果が得られた．3 つの抽象化の中で，繰り返しパターンが最も一致率が高く，偶然一致してしまう可能性がある．繰り返しパターンは，その入れ子の形状の情報も併せて持つため，入れ子の内部の情報が異なっていても入れ子の形状の一致する場合があるためと考えられる．

4.2 仕様が異なる場合

一般的に仕様が異なることで盗用部分の実行方法も異なり，同一のメソッドを実行しても得られる実行系列は変化する．本来，盗用関係にあるはずのメソッド同士であっても，その実行方法の違いによってバースマークが変化してしまい盗用の発見が困難になると考えるのが一般的である．

4.2.1 保存性の確認実験

盗用状況として，Jakarta のライブラリ L_1 を利用し実装者 A に仕様 1 のプログラム $P_{(L_1,A)}$ を実装させる．次に，実装者 B に L_1 を用いて仕様 1 とは異なる仕様 2 [*6]のプログラム $Q_{(L_1,B)}$ を実装させる．$Q_{(L_1,B)}$ はライブラリ L_1 を盗用して実装された盗用プログラムである．

比較結果から一部抜粋したものを**表 3** に示す．

表 3　異仕様における抽象表現の保存性確認

	抽象化 1	抽象化 2	抽象化 3
改変なし	0.76	0.74	0.79
ArraySplitter	0.29	0.20	0.15
BranchInverter	0.76	0.74	0.79
ClassSplitter	0.59	0.55	0.63
ConstantPoolReorderer	0.76	0.74	0.79
PromotePrimitiveTypes	0.14	0.03	0.04
ReorderParameters	0.76	0.74	0.79
SimpleOpaquePredicates	0.76	0.74	0.79

表 1 の同一仕様による実験結果と比べ全体的に一致率

[*2] 本稿では特に，一般的なプログラムの実装が容易であることから正規表現を扱うライブラリを利用した．
[*3] 約 28KB，クラス数 15，メソッド数約 120
[*4] 指定正規表現にマッチした文字列を指定文字列で置換する．
[*5] 約 93KB，クラス数 14，メソッド数約 70
[*6] 指定正規表現にマッチした文字列を区切りにトークンに分割する．

が 0.2 ポイント程度下がる傾向にある．また同一仕様の実験結果と同様に多くの難読化アルゴリズムに対して耐性を示したが，やはり標準 API 呼び出しが増えるような難読化に対しては耐性を得られなかった．

4.2.2 弁別性の確認

弁別性を確認するために，GNU のライブラリ L_2 を用いて実装者 B に仕様 2 のプログラム $Q_{(L_2,B)}$ を実装させる．$P_{(L_1,A)}$，$Q_{(L_2,B)}$ は完全に独立で実装された非盗用関係である．$P_{(L_1,A)}$ と $Q_{(L_2,B)}$ の一致率の結果から一部抜粋したものを**表 4** に示す．

表 4　異仕様における抽象表現の弁別性確認

	抽象化 1	抽象化 2	抽象化 3
改変なし	0.33	0.03	0.10
ArraySplitter	0.18	0.05	0.05
BranchInverter	0.31	0.03	0.10
ClassSplitter	0.22	0.03	0.06
ConstantPoolReorderer	0.31	0.03	0.10
PromotePrimitiveTypes	0.11	0.01	0.01
ReorderParameters	0.31	0.03	0.10
SimpleOpaquePredicates	0.31	0.03	0.10

同一仕様の場合とほぼ同一の結果が得られた．全体的に保存性の確認実験と比べ一致率が半分以下となった．また，出現間隔と近傍集合の一致率は保存性と比べ極端に低い結果が得られた．

5　実験結果の考察

同一仕様における表 1，表 2 の実験結果から，一部の攻撃には耐性を示すことができなかった．しかし，それ以外のほとんどの攻撃に対しては影響を受けておらず，特に繰り返しパターンと近傍集合による抽象化を行った場合は 9 割以上の一致率を示し保存性を十分に満たしたと言える．耐性を持たなかった攻撃は，標準 API メソッドの呼び出しの追加がされる手法であるため，得られる実行系列が変化した結果であると考えられる．

非盗用関係における一致率は最大繰り返しパターンの 3 割程度となり，弁別性を十分に満たすと言える．繰り返しパターンが他と一致率が高い原因として，繰り返しの入れ子の中の実行順序などの詳細な情報が異なっていても，繰り返しの入れ子の形状が一致することが考えられる．

また，異仕様に実装した場合には表 3，表 4 の結果が得られた．同一仕様の場合と比べて，全体的に保存性を示す一致率は 0.2 ポイント程度低下し，弁別性を示す一致率には大きな変化がなかった．保存性の低下は，仕様によって盗用部分の実行方法が変化し，それによって実行系列が変化したことが原因であると考えられる．しかし，それでも一致率は 7 割以上を示しているため，実行系列の変化に対してもそれぞれバースマークは耐性を持つと言える．弁別性の実験結果は同一仕様の場合とほぼ同じであったため，特に繰り返しパターンの保存性と弁別性の値が近くなっている．しかし，保存性と弁別性には 2 倍近い差があり，保存性・弁別性を示すのに十分であると考えられる．

文献 [3] による弁別性の実験において，同一仕様のプログラムを異なる実装者に実装させた場合，実装に用いるクラスなどが偶然一致してもプログラム全体のバースマークが弁別性を示す結果が得られている．つまり従来の研究では，同一仕様に実装される部分盗用の状況において，盗用部分の保存性を示すことができなかった．しかし本研究では，部分系列の抽出を行うことでプログラム全体では判別できなかった盗用部分と盗用者による実装部分の判別が可能となった．さらに抽象化によって盗用部分で定義されるメソッドの実行順序情報を除くことで，実装者よって異なる実行方法によって変化してしまう実行系列から本質的な情報を得ることができた．その結果，想定する盗用状況において盗用部分の保存性・弁別性を示す結果が得られ，バースマークの部分盗用発見への有効性を示したと言える．

6　おわりに

本研究では，[4]，[3] によるバースマーク構成の枠組みを部分盗用の状況に合わせて拡張する方法について検討を行った．その結果，多くの部分盗用の状況に対して保存性・弁別性を示すことができ，従来のバースマークが発見できなかった部分盗用への有効性を示すことできたと言える．

しかし，標準 API メソッドの呼び出しが増える一部の攻撃には保存性・弁別性を示すことができなかった．今後の課題として，より効果的に部分盗用へ応用するために，盗用単位やプログラム仕様に依存しない実行系列，抽出系列，部分系列について検討し，プログラムの持つ本質的な特徴を抜き出す方法について検討する必要がある．

参考文献

[1] Ginger Myles et al.：“ Detecting Software Theft via Whole Program Path Birthmarks ”, ISC2004 in LNCS 3225，pp.404-415 (2004).

[2] 岡本ら：“ API 呼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 D, Vol.J89-D, No.8, pp.1751-1763 (Aug.2006).

[3] 林ら：“ 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 ”，情報処理学会論文誌，Vol.48, No.4 (Apr.2007)(掲載決定).

[4] 古田ら：“ 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-D-I No.10，pp.1595-1598 (Oct.2005).

[5] 森山ら：“ API 関数呼出履歴によるソフトウェア動的バースマークの一方式 ”，電子情報通信学会技術研究報告 ISEC2006-82，pp.77-84 (Sep.2006).