社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2003－ＣＳＥＣ－22　(18)
2003／7／17

Javaクラスファイルからプログラム指紋を抽出する方法の提案

玉田　春昭　　神崎雄一郎　　中村　匡秀　　門田　暁人　　松本　健一

奈良先端科学技術大学院大学 情報科学研究科　〒630-0101 奈良県生駒市高山町 8916-5
E-mail: {harua-t,yuichi-k,masa-n,akito-m,matsumoto}@is.aist-nara.ac.jp

あらまし　本論文では盗用の疑いのあるJavaプログラムの発見を容易にすることを目的として，Javaクラスファイルからプログラム指紋を抽出する方法を提案する．提案方法はプログラム中の特徴的な箇所である初期値代入部分，メソッド呼び出し部分などを抽出し，指紋として用いる．このプログラム指紋を用いることにより，Javaクラスファイルを互いに区別することが可能となる．検証実験において，J2SDK SE 1.4.1_02のクラスライブラリに適用した結果，提案手法により99.94%のクラスを互いに区別できることを確認した．

キーワード　プログラム指紋, ソフトウェアセキュリティ, Java, ソフトウェア識別

A method for extracting program fingerprints from Java class files

Haruaki TAMADA, Yuichiro KANZAKI, Masahide NAKAMURA, Akito MONDEN, and Ken-ichi MATSUMOTO

Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5 Takayama-cho Ikoma-shi, Nara, 630-0101 Japan
E-mail: {harua-t,yuichi-k,masa-n,akito-m,matsumoto}@is.aist-nara.ac.jp

Abstract　To support efficient detection of illegal theft for Java class files. This paper presents a new method to derive "program fingerprints" from given Java class files. For a given class file, the proposed method extracts unique property from the class file based on initial value assignments, the sequence of method calls and the inheritance structure. By using the fingerprints, any Java class file can be easily distinguished from other class files. We evalute the proposed method by applying it to J2SDK SE 1.4.1_02. The experimental result shows that the proposed fingerprints can identify 99.94% of all the class files involved.

Key words　program fingerprints, software security, Java, software identification

1. はじめに

近年，ソフトウェアのライセンスに関わる様々な事件が起こっている[12][11][13]．例えば，オープンソースソフトウェアのライセンス形態であるGPL[8]を適用しているソフトウェアを，内部に組み込んでいるにも関わらず，ソースファイルを公開せずに(注1)，ソフトウェアを発表したり，元のソフトウェアのライセンスに違反してそのソフトウェアを組み込んだ新しいソフトウェアを発表したという事件である．これらの事件ではライセンスに違反した，いわばソフトウェアの盗用である．これらの事件では，ソフトウェアの盗用を発見，証明するためにバイナリの中から，盗用されたソフトウェア特有のコードが探され，バイナリ中から元のソフトウェアのCopyright文が発見されたことが盗用を証明する足掛かりとなった．このバイナリ中から盗用されたソフトウェア特有のコードを探すという作業は知識のある者の手が不可欠だが，時間もかかり，確実に証明できるものではない．更に，その証明を知識のない者に理解させることは大変困難である．

一方，近年Javaが携帯電話から基幹アプリケーションまで様々な分野で使われてきている．Javaで作られたアプリケーションは，必ずプログラムの単位である複数のクラスファイルから構成される．この時，クラスファイルのいくつかを無断で別のアプリケーションに盗用することは技術的に容易である．また，標準APIに含まれていないような汎用的なクラスファイル，例えばBase64の入出力ストリームのクラスファイルなどであれば，より一層盗用の危険性は増すであろう．加えてJavaにはJad[9]をはじめとする逆コンパイラが他の言語に比べて普

(注1)：GPLであるソフトウェアを組み込んだ場合，同じGPLでソフトウェアを公開しなければならない

及しており，クラスファイルからソースファイルに戻すことが可能となっているため，Javaクラスファイルは盗用の危険性が他の言語と比べて高くなっている．

そこで，本稿では盗用の事実発見を容易にすることを目的としてJavaクラスファイルからプログラム指紋を抽出することで，Javaクラスファイルを互いに識別する方法を提案する．ここで言うプログラム指紋とはクラスファイルに含まれるそのクラス特有の特徴のことを指す．実行に不可欠な部分であるため，取り外すことが不可能である．このプログラム指紋により，盗用の疑いのあるクラスファイルとオリジナルのクラスファイルが同一のクラスファイルであるかどうかを判定でき，盗用の事実を証明することができる．また，提案手法をJ2SDK SE 1.4.1_02のクラスファイルに適用した結果，99.94%のクラスを互いに区別できることが確認できた．

2. 関連研究

ソフトウェアの盗用を証明する方法として，従来より**透かし**を埋め込む方法がいくつか提案されている[6][7][1][2].

北川ら[6]は配列に数値列を透かしとして埋め込む方法を提案している．この手法は配列変数を用意し，その配列変数に透かし情報となる数値列を代入することで透かしを実現している．

門田ら[7]はJavaプログラムにダミーメソッドを追加し，そのダミーメソッド内の数値オペランド，オペコードを置き換え，透かしを埋め込んでいる．

プログラムの盗用が発生した時，これら透かしを取り出すことでソフトウェアの所有権を主張することができるが，これらの方法には2つの問題点があることが知られている．

1つは攻撃者に透かしを外されてしまう危険性があるということである．ソフトウェアの透かし情報はプログラム実行の面では不必要なものであり，知識のある攻撃者であれば透かし情報が埋め込まれている場所を特定することも場合によっては可能である．加えて，今日Javaクラスファイルに対する処理を行うためのライブラリやツールが数多く発表されており，透かしを埋め込んだ箇所を特定することがより容易になっている．透かしが外されてしまうと，透かしではクラスファイルを識別することができなくなってしまうため，盗用の証明が困難になる．

もう1つの問題点は，識別対象が透かしの埋め込まれているソフトウェアに限られることだ．予め透かしを埋め込んでおかなければならない電子透かしの性質上，透かしが埋め込まれていないソフトウェアの盗用を証明することは不可能である．

以上のようにプログラムの静的な部分に透かしを埋め込む方法とは異なり，Collbergらはプログラムの動的な部分に透かしを埋め込む方法を提案しており，それらは以下の3つに分類される[1][2].

イースターエッグ　あるユニークな入力Iに対して透かし情報が出力される．

データ　あるユニークな入力Iに対してデータに透かしが現れる．

実行トレース　あるユニークな入力Iに対してプログラムの実行トレースに透かしが現れる．

動的な透かしは静的な透かしに比べ，難読化や最適化などのプログラムの等価変換に対しても強い耐性を持つ．しかし，これらの方法でも，予め透かしを埋め込んでおかなければならず，攻撃者に透かしを外される危険性を解決するには至っていない．

このように，透かしを埋め込むことでソフトウェアの盗用を証明する方法は問題点を有しており，盗用を証明する手段としては完全ではない．

3. プログラム指紋

従来，ソフトウェアの指紋(fingerprint)というと狭義に透かしのことを指すことがあるが，本稿では厳密には違う概念を指すことにする．具体的にはプログラム指紋を以下のように定義する．

[定義] Pを与えられたプログラムとし，Pから何らかの方法fで抽出される特徴の集合を，$f(P)$とする．$p = f(P)$が以下の条件を満たすとき，pをPのプログラム指紋という．

条件C1 Pからpを取り外すことは不可能である．

条件C2 あるプログラムQに対して，$p \in f(Q)$ならば，$P \not\equiv Q$である．ここで$\equiv$は，プログラムの与えられた等価関係とする．

条件C1について，取り外すことは不可能であるということは，プログラムの付加的な情報ではないということである．つまり，仕様通りにプログラムが動作するために不可欠な情報のことを指す．このことから透かしのように予め情報を埋め込むことはない．

条件C2における等価関係$\equiv$は，直感的には，2つのプログラムPとQが全く同一かあるいは盗用(コピー)である場合，$P \equiv Q$が成り立つと解釈する($P \equiv P$は常に成立)．盗用であると判断する基準は，様々なものが考えられるため，ここでは特に具体的な定義は行わず，与えられるものと仮定している．さらに，PとQの機能的な仕様を$Spec(P), Spec(Q)$とする時，$P \equiv Q \Rightarrow Spec(P) = Spec(Q)$が成り立つものとする(逆は必ずしも成り立たない)．

例えば，Pの識別子名だけをある方法で付け替えたプログラムQを$P \equiv Q$と定義すると，Javaプログラム上の機能的な仕様は同じとなる．しかし，逆は必ずしも成立しない．P'とQ'の仕様が同じであっても，Q'は必ずしもP'の名前付け替えによって得られたものとは限らない．

条件C2では，もしPとQから抽出された指紋$f(P)$と$f(Q)$とを比較して異なっていれば，$P \not\equiv Q$，つまり，QはPの盗用でないことを保障している．

補足として，指紋の集合もまた指紋となるため，異なる種類の指紋を一つの指紋として扱える．

また，プログラム指紋には以下の性質も併せて満たすことが望ましい．

条件D1 Pを難読化あるいは等価変換したP'からはpが得られる．

条件D2 異なる人が独自に作成した同じ仕様のプログラムPとP'から得られる指紋はpとp'であり，$p \neq p'$である．

盗用者にとっては，プログラムの盗用が発覚しないよう，盗用するプログラムに対して何らかの手を加えることが多いと考えられる．この手を加える方法の1つとして難読化が使われることもあるであろう．難読化は本来プログラムを保護する目的のため使われるものであるが，悪用することで攻撃にも使うことができる．このような難読化が行われたプログラムであっても同じ指紋が抽出できることが望ましい．しかし，難読化手法は今日様々な方法が提案されており，それら全ての難読化手法に対応するとなると取り出せる特徴が少なくなり，プログラム指紋の条件C2を満たさなくなる可能性もある．

また，同じ仕様のプログラムを全く独自に作成した場合であっても，内部の処理までもが全く同一であれば，抽出される指紋は同じものである．規模の大きなプログラムであれば細部まで同じということは少ないだろうが，小さなプログラムであれば一致してしまう偶然も存在し得る．これらの性質については許容範囲を適宜設定する必要がある．

4. 提案手法

4.1 構　　成

提案する手法の基本構成図を図1に示す．

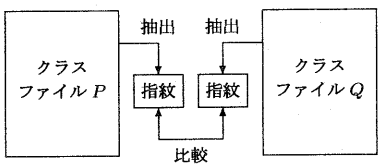

図1　基本構成

2つのクラスファイルから同じ種類の指紋を抽出し，抽出された指紋同士を比較して，完全に一致すれば二つのプログラムは同一であると判断する．抽出する指紋は**初期値代入部分**，**メソッド呼び出し**，**継承関係**の3つに着目する．

これらの情報を指紋として抽出できないクラスも存在するが，比較する2つのプログラムどちらからも抽出できない場合は無視し，一方からしか抽出できない場合は一致しないものとする．

この指紋として扱うものの中にはユーザ定義のメソッド名やフィールド名，クラス名は含めない．なぜなら，今日流通している難読化ツールの代表的な処理が**名称の変更**であるためであり，Javaのバイトコードを扱うライブラリを使用すると容易に変更可能な情報であるためである．ここで言うユーザ定義のクラスとは標準API，標準拡張API，有名団体から提供されているAPI以外のことを指し，パッケージ名が `java`, `javax`, `com.sun`, `sun`, `com.apple`, `org.apache`, `org.omg`, `org.xml`, `org.w3c`, `org.ietf`, `junit`, `org.gnu`, `pnuts`, `org.eclipse` から始まっていないパッケージに所属するクラスを指す．これらのパッケージはJavaが動くために必要であったり，各ユーザが独自に入手していたりするため，ユーザが直接変更できるようなものではない．そのため，変更するとプログラム自体が動かなくなる恐れもあり，指紋として扱える．

これら指紋を抽出する対象はクラスに限定し，インターフェースは対象としない．なぜなら，インターフェースはメソッドの中身が存在しないので，アルゴリズムなどを保護する必要はないためである．

以下にそれぞれの指紋の抽出方法を述べる．

4.2 各指紋の抽出方法

4.2.1 初期値代入

クラスのフィールドへの初期値代入部から指紋を抽出する．

初期値代入部分の指紋としての扱いは，プリミティブ型と `java.lang.String` 型に限定する．2つのクラスファイル P, Q の初期値代入において，その型と値が一致すれば指紋として一致するものとする．しかし，フィールドの中には初期値が代入されないものやコンストラクタの引数で決定される場合もある．このような場合はそのフィールドは初期値なしとして扱う．

初期値が代入されているかどうかはstaticイニシャライザ，コンストラクタ，そして，クラスファイルに含まれる `ConstantValue` 属性のいずれかを見ることでわかるようになっている．

`static final` 宣言されているフィールドの値はクラスファイルの`ConstantValue` 属性に初期値が格納されている．`static` 宣言されているフィールドの値はstaticイニシャライザで代入が行われ，そうでないフィールドはコンストラクタで代入が行われる．これらを調べ，型と値のペアをそのクラスの初期値代入指紋として扱う．

具体的な取得方法を図2に示す．

まずクラスファイルにおけるフィールドを一つずつ調べる．もし，`ConstantValue` 属性があれば，その値を初期値とする．そうでなければstaticフィールドであるかどうかを調べる．staticの場合，初期値代入は *putstatic* というインストラクションで行われる．一方，一般のフィールドであれば *putfield* というインストラクションによって初期値が行われる．

putstatic, *putfield* インストラクションを実行するにはスタックに代入される値が積まれている必要がある．その値が静的に決まる場合は *ldc*, *iconst*, *lconst*, *fconst*, *dconst*, *bipush*, *sipush* のいずれかのインストラクションが *putstatic*, *putfield* の前に現れている．これらを見つけ出し，初期値が代入されていると判断する．

具体的な取得手順を以下のクラスを例に挙げて説明する．

```
public class SampleClass{
  public static int value1 = 10;
  private int value2;
}
```

このクラスをコンパイルしてできたクラスファイルを逆アセンブルすると以下の出力が得られる．

```
Compiled from SampleClass.java
public class SampleClass
        extends java.lang.Object {
  public static int value1;
  private int value2;
  public SampleClass();
  static {};
}

Method SampleClass()
 0 aload_0
 1 invokespecial #1
     <Method java.lang.Object()>
 4 return

Method static {}
 0 bipush 10
 2 putstatic #2 <Field int value1>
 5 return
```

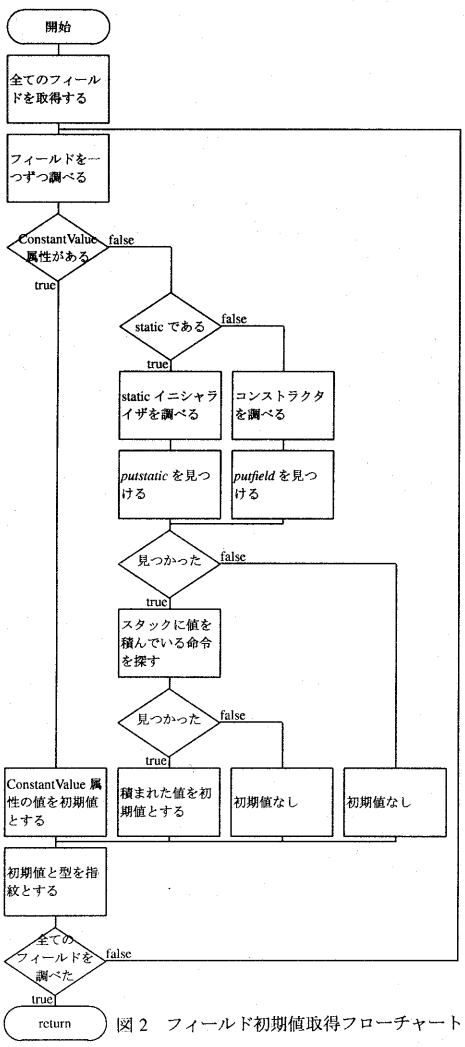

図 2　フィールド初期値取得フローチャート

上記のバイトコードから以下の手順により初期値代入の指紋を抽出する．

（1） SampleClass のフィールドは int 型の変数 value1 と value2 である．

（2） value1 は static 変数なので，static イニシャライザを調べる．

（3） static イニシャライザの中に *putstatic* が見つかる．

（4） *putstatic* の前に *bipush* が見つかる．

（5） value1 の初期値は 10 である．

（6） value2 は static 変数ではないので，コンストラクタを調べる．

（7） コンストラクタの中に *putfield* が存在しない．

（8） value2 の初期値はない．

以上の操作により `SampleClass` の初期値代入指紋 (int 10), (int -) が得られる．

4.2.2　メソッド呼び出し

ユーザ定義でないクラスのメソッド呼び出しと順番を指紋として扱う．

調べる対象のクラス P の各メソッドから全てのユーザ定義でないメソッド呼び出しを調べ，比較対象のクラス Q からも同様に全てのユーザ定義でないメソッド呼び出しを調べる．その後，P と Q の各メソッド毎にメソッド呼び出しの順と呼び出されているメソッドが一致しているかを調べ，全てが一致していれば同じメソッドであると判断する．これを全てのメソッドに対して調べる．

この時，メソッド名の一致は調べないが例外として，static イニシャライザ，コンストラクタ，main メソッドの 3 つのメソッドのみはメソッド名も一致させる．コンストラクタ，static イニシャライザはコンパイルされるとそれぞれ，`<init>`，`<clinit>` という名前になる。また、main メソッドはプログラムが起動する時に呼び出されるメソッドである．そのため，これらのメソッド名は変更してしまうとプログラムが仕様通り動かなくなってしまうので，名称を変更する難読化ツールであっても，これらのメソッド名を変更することはない．従って，これらの情報も指紋として扱えることになる．また，abstract メソッド，native メソッドはメソッドの中身が存在しないため，無視する．これを例を用いて説明すると以下の通りである．

例えば，以下のようなメソッドをコンパイルすると

```
public String sampleMethod(){
  return getClass().getName() +
      "#sampleMethod()";
}
```

以下のバイトコードに変換される．

```
 0 new #2 <Class java.lang.StringBuffer>
 3 dup
 4 aload_0
 5 invokevirtual #3 <Method
     java.lang.Class getClass()>
 8 invokevirtual #4 <Method
     java.lang.String getName()>
11 invokespecial #5 <Method
     java.lang.StringBuffer(
         java.lang.String)>
14 ldc #6 <String "#sampleMethod()">
16 invokevirtual #7 <Method
     java.lang.StringBuffer
       append(java.lang.String)>
19 invokevirtual #8 <Method
     java.lang.String toString()>
22 areturn
```

ここで指紋となるのは斜体で書かれたものとその順番である．ソースコードにおいては2行の簡単なプログラムであるが，この2行の中で5回のメソッド呼び出しを行っている．1つ目は getClass() というメソッド呼び出しであり，2つ目は getName() というメソッド呼び出しである．3つ目のメソッド呼び出しはソースコード中には出てこないが，文字列を + 記号で連結しているため，StringBuffer クラスのコンストラクタが呼び出されている部分となる．そして，4つ目のメソッド呼び出しは "#sampleMethod()" という文字列を先ほど構築した StringBuffer に連結するための append(String) となる．最後の5つ目は StringBuffer を String に変換するための toString() というメソッド呼び出しである．これらの呼び出し順序が一致していれば指紋が同じであると言える．また，sampleMethod というメソッド名は指紋としては扱わない．

この場合，sampleMethod から抽出されるメソッド呼び出し指紋は(invokevirtual Object#getClass()), (invokevirtual Class#getName()), (invokespecial StringBuffer#<init>), (invokevirtual StringBuffer#append(String)), (invokevirtual StringBuffer#toString()) となる．

4.2.3 継承関係

これは継承の関係を指紋として扱う．対象のクラスのスーパークラスを順に辿っていき，ルートまでの数と，辿っていったスーパークラスがユーザ定義のクラスでない場合はそのクラス名が一致するかどうかを調べる．ユーザ定義のクラスの場合はクラス名が一致するかどうかは調べない．継承の深さと，ユーザ定義でないスーパークラスの名前が全て一致した場合に指紋が合致したとする．

以下の図3に表すクラスの継承関係を例に挙げ，継承関係の指紋について説明する．

MyResource_ja と MyResource_en は継承の深さと継承しているクラスも同一であるため，同じ指紋であると判断する．MyResource_ja_JP と MyResource_en とは継承の深さが異なっているため，異なる指紋である．MyResource_ja_JP と MyResource_en_CA は継承の深さは同じであるものの，スーパークラスが異なっている．しかし，これらのクラスのスーパークラスはユーザ定義のクラスである．ユーザ定義のクラスの名称は指紋としては使わないので，MyResource_ja と MyResource_en のクラス名の比較は行わず，MyResource_ja_JP と MyResource_en_CA の継承関係の指紋は一致する．すなわち，両者の指紋はともに (-/ListResourceBundle/ResourceBundle/Object) となる (- はドントケアを表す)．

5. ケーススタディ

以下に3つの指紋でクラスファイルを識別する例を挙げる．

5.1 ソースファイル

π を求める3つのプログラムを対象にプログラム指紋を比較する．それぞれ，乱数を用いたモンテカルロ法，arctan を用いた J.Machin の公式，相加相乗平均を用いた方法により π を求めている．それぞれのソースファイルを以下に載せる．

```
public class MonteCarloPi{
  private static final int LOOP_COUNT
      = 100000;
  public double pi(){
    int hit = 0;
    for(int i = 0; i < LOOP_COUNT; i++){
      double x = Math.random();
      double y = Math.random();
      if(x * x + y * y < 1) hit++;
    }
    return 4d * hit / LOOP_COUNT;
  }
} // モンテカルロ法
```

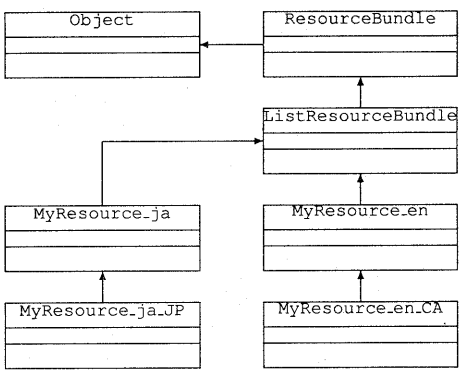

図3　継承関係

```
public class ArctanPi{
  public double pi(){
    return 4 * (4 * Math.atan(1d/5d) -
                Math.atan(1d/239d));
  }
} // J.Machin の公式
```

```
public class AgmPi{
  public double pi(){
    double a = 1;
    double b = 1 / Math.sqrt(2);
    double s = 1;
    double t = 4;
    double last, result = 0d;
    for(int i = 0; i < 3; i++){
      last = a; a = (a + b) / 2;
      b = Math.sqrt(last * b);
      s -= t * (a - last) * (a - last);
      t *= 2;
      result = (a + b) * (a + b) / s;
    }
    return result;
  }
} // 相加相乗平均
```

5.2 指紋の比較

上記の3つのクラスに対して指紋を抽出し，比較する．抽出した指紋を表1に載せる．

表1 指紋の比較

	MonteCarloPi	ArctanPi	AgmPi
初期値代入	int/100000	なし	なし
メソッド呼び出し	Math.random Math.random	Math.atan Math.atan	Math.sqrt Math.sqrt
継承関係	1/Object	1/Object	1/Object

初期値代入の行は，型と初期値を / で区切っている．また，継承関係の行についても，継承の深さとスーパークラス名を / で区切っている．

初期値代入の指紋は ArctanPi，AgmPi クラスの2つはフィールドが存在しないため，この指紋を取り出すことはできない．MonteCarloPi クラスは LOOP_COUNT という定数があり，int 型の 100000 という値であり，これが指紋となる．この指紋により MonteCarloPi クラスは他のクラスと異なることがわかる．継承関係はいずれのクラスも同じく Object クラスを継承しているため，この指紋では区別することができない．そして，メソッド呼び出しの指紋においては3つのクラスが全て異なるメソッドを呼び出している．このことから ArctanPi と AgmPi も指紋が異なるとわかる．以上のような指紋の比較により，3つのクラスが異なる指紋を持ち，異なるプログラムであることがわかる．

6. 評　　価

以上の手順により指紋の抽出，および比較を行うツールを作成し，実験を行った．Windows 用の Java 2 SDK Standard Edition 1.4.1_02 の rt.jar に含まれる 8,219 のクラスファイルの内，7,249 のクラスに対して2つずつ相互に上で述べた3つの指紋で比較を行った．

26,270,376 回の比較を行い，指紋が一致した回数はその内 13,404 回となり，99.94% の確率で区別できることがわかった．この結果を表2に載せる．

表2 結果

	rt.jar
対象クラス数	7,249
比較回数	26,270,376
一致回数	13,404
区別できた割合	99.94%

表2において区別できた割合は一致回数を比較回数で割り，その結果を1から引くことで計算している．

偶然にも指紋が一致したクラスは，クラス内でほとんど処理を行っていないものが多く，クラスファイルのサイズも小さいものが多い．

例えば指紋が偶然一致したクラスの中には Adapter クラスや，java.util.ListResourceBundle のサブクラスが存在する．Adapter クラスはサブクラス化されることが前提となっており，メソッドの中身は何もないことが多い．従って提案した指紋では区別できない．

また，ListResourceBundle のサブクラスはコンストラクタと Object[][] getContents() というメソッドのみを定義すれば良く，ResourceBundle というクラスはプログラムのリソースを管理するために存在するものなのでほとんどの場合，getContents が返すオブジェクトは変化しない．そのため，そのオブジェクトは static な定数として持っていることが多く static イニシャライザでリソースの初期化を行っているため，メソッド呼び出しの数が少なくなっている．また，これらのリソースクラスは国際化のために表示文字列を様々な言語で表現できるように全く同一の構造になっている．

このようにクラス同士の指紋を比較して偶然一致したものはクラス内であまり処理を行わないものが多く，メソッド呼び出しの指紋では区別できなかったものと思われる．継承関係の指紋も，ほとんどが java.lang.Object を継承しており，この指紋でも区別できない．初期値の指紋についても抽出できないクラスも存在する．これらの理由のため，指紋で区別できなかったと考えられる．

7. まとめと今後の課題

本稿では Java クラスファイルの盗用を容易に証明するために，Java クラスファイルからクラスファイル特有の特徴を取り出して指紋とすることで，クラスファイルを区別する方法を提案した．指紋がクラスファイル同士を区別できるか否かを実験

により評価した結果，偶然一致する場合があるものの，ほとんどの場合，区別できることが確認された．

従来研究においては予めソフトウェアに情報を透かしとして埋め込んでおき，盗用を証明する際には透かしを取り出すという方法が論じられているが，透かしが埋め込まれていないソフトウェアの盗用を証明する方法は述べられていない．

これに対して，本稿では透かしが予め埋め込まれていないという前提で盗用されたJavaクラスファイルの発見という目的の解の提案を行った．提案方法を用いることによって，どのようなJavaのクラスファイルであっても，容易に盗用の事実を証明することが可能になる．

現在のプログラム指紋はクラス内で処理をあまり行っていないクラスの場合に，偶然一致してしまうという問題がある．また，難読化などの攻撃に対する耐性については言及していない．今後は，本稿で区別できなかったプログラムであっても区別が行えるよう，クラス間の関係を指紋として扱うべく検討していく予定である．また，難読化されたプログラムからも同じ指紋が取得できるよう，模索していく予定である．

文　献

[1] Christian Collberg and Clark Thomborson, "Software Watermarking: Models and Dynamic Embeddings," In Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principle Languages (POPL'99), San Antonio, Texas, Jan. 1999.

[2] Christian Collberg and Clark Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection," IEEE Transactions on Software Engineering 28:8, 735-746, Aug, 2002.

[3] 福島和英, 櫻井幸一, "ソフトウェア透かしにおける個人識別情報埋め込み位置の難読化", 暗号と情報セキュリティシンポジウム (SCIS2003), pp.1053-1058, Jan, 2003

[4] 福島和英, 櫻井幸一, "メソッド分散による Java 言語の難読化手法の提案", コンピュータセキュリティシンポジウム 2002 (CSS2002), pp.191-196, Oct, 2002

[5] Josef Pieprzyk, "Fingerprints for copyright software protection." In Masahiro Mambo and Yuliang Zheng, editors, Proceedings of the 2nd Information Security Workshop (ISW'99), volume 1729 of LNCS , pages 178-190. Springer, Nov, 1999.

[6] 北川隆, 椹勇一, 嵩忠雄, "Java で記述されたプログラムに対する電子透かし法", 暗号と情報セキュリティシンポジウム (SCI 1998),

[7] 門田暁人，松本健一，飯田元，井上克郎，鳥居宏次, "Java クラスファイルに対する電子透かし法". 情報処理学会論文誌, Vol. 41 No.11, Nov, 2000

[8] GNU General Public License
http://www.gnu.org/copyleft/gpl.html

[9] jad - the fast java decompiler
http://kpdus.tripod.com/jad.html

[10] jmark: A lightweight tool for watermarking Java class files
http://se.aist-nara.ac.jp/jmark/

[11] 「ネットげーせん」(事実上無期) 延期
http://slashdot.jp/article.pl?sid=02/09/17/1446214

[12] オールドゲーム配信の裏にライセンス違反?
http://slashdot.jp/article.pl?sid=02/08/28/0423226

[13] プロジーの DivX コンバータに GPL 違反の疑い
http://slashdot.jp/article.pl?sid=03/02/05/1738259