# クラス間におけるフィールドおよびメソッドのアクセス関係に着目したJavaソースコードの難読化手法

福島, 和英
九州大学大学院システム情報科学府

田端, 利宏
九州大学大学院システム情報科学研究院

櫻井, 幸一
九州大学大学院システム情報科学研究院

https://doi.org/10.15017/4783634

出版情報：九州大学情報基盤センター年報. 4, pp.31-39, 2004-03. 九州大学情報基盤センター
バージョン：
権利関係：

情報基盤センター年報第4号（2004年）
九州大学情報基盤センター

# クラス間におけるフィールドおよびメソッドのアクセス関係に着目したJavaソースコードの難読化手法

# Obfuscation Scheme for Java focusing on Access Relation between Classes

福島 和英† 田端 利宏‡ 櫻井 幸一‡
Kazuhide Fukushima † Toshihiro Tabata ‡ Kouichi Sakurai ‡

† … 九州大学大学院システム情報科学府
† … Graduate School of Information Science and Electrical Engineering, Kyushu University
‡ … 九州大学大学院システム情報科学研究院
‡ … Faculty of Information Science and Electrical Engineering, Kyushu University

**要旨** 近年，高い移植性を持つJavaが広く普及している．しかし，Javaには攻撃者が実行可能ファイルであるクラスファイルからソースコードを容易に復元できるという問題点がある．このため，Javaプログラムに対する耐タンパー化技術の研究が幅広く行われている．しかし，耐タンパー化技術の安全性に対する定量的な評価は，十分に与えられていないのが現状である．本論文では，カプセル化構造の破壊に着目したJavaソースコードに対する難読化手法を提案する．さらに，複数のクラス間にまたがるメソッドとフィールドの呼び出し回数に基づく難読化の評価尺度を提案する．この尺度により，一定の定量的根拠を持つ耐タンパーソフトウェアの実現を試みる．

**Abstract** Recently, Java has been spread widely. However, Java has a problem that an attacker can reconstruct Java source codes from Java classfiles. Therefore many techniques for protecting Java software have been proposed, but, quantitive security evaluations are not fully given. This paper introduces an obfuscation scheme for Java source codes by destructing the encapsulation. Moreover, we give an evaluation measure based on the number of accesses to the fields and the methods of another class. We try to realize of tamper-registrant software with the certain quantitive basis of security using our mesure.

## 1 はじめに

近年，技術の向上によってハードウェアの費用が劇的に低くなっている．一方，ソフトウェアの費用は相対的に高くなっている．このことは，計算機性能の向上により複雑な処理を実行することが可能となったことに伴い，開発されるソフトウェアも複雑で規模が大きいものになった点に起因している．また，現在の技術をもってしても，ソフトウェアの作成は人手に頼る面が大きいという点も大きな要因である．

ソフトウェアがネットワークを介して広範囲に流通すると，ソフトウェアの著作権保護を考慮する必要がある．不正者によるソフトウェア内のデータや鍵となるアルゴリズムの盗用からの保護がこれに当てはまる．ソフトウェアの開発には大きな工数を要する．しかし，不正者が容易にソフトウェアを盗用するという状況が起こりうる．このため，ソフトウェアを保護するための情報セキュリティ技術が必要である．

近年，Javaが急速に普及している．携帯電話やPDAなどの小型情報端末上でも実行できるという高い移植性や多くの人々が関心を持っているインターネットとの高い親和性が大きな要因に挙げられる．しかし，Javaは著作権保護が特に重要である．Javaでは，仮想マシン上で実行されるクラスファイルが一般的な流通形態である．このクラスファイル中には，ソフトウェア構造を知るための手掛かりとなりうるクラス名，スーパークラス名，メソッド名およびフィールド名などの情報がある．さらに，クラスファイルの記述はフィー

ルドやメソッド単位に分かれている [6]．このため，クラスファイルの可読性は高く，高性能の逆コンパイラが開発されている．この結果，攻撃者は，クラスファイルを逆コンパイルすることによって，簡単に元のソースコードに近いソースコードを入手できる．さらに，攻撃者が得られたソースコードに対してリバースエンジニアリング（逆行解析）を行うことにより，ソフトウェア中の秘密のデータやアルゴリズムを盗用できる可能性がある．

この問題を解決するための一つの手法としてソフトウェアの耐タンパー化技術がある．耐タンパーソフトウェアとは，外部からの観測や不当な改ざんを行うことが困難なソフトウェアのことである．耐タンパー性を実現するため，現在までにさまざまな手法が提案されている．

本論文では，カプセル化構造の破壊に着目した Java ソースコードに対する難読化手法を提案する．さらに，メソッドやフィールドの複数のクラス間にまたがる呼び出し回数に基づく難読化の評価尺度を提案する．

## 2 関連研究

ソフトウェアの著作権保護手法は大きく 2 つに分類できる．1 つは，ソフトウェアに対する外部からの観測，改ざんを難しくすることにより，ソフトウェア内に含まれる秘密のデータや価値のあるアルゴリズムなどの知的財産を保護する手法である．もう 1 つは，何らかの手法によりソフトウェアの同一性を証明する手法である．この手法を用いることにより，ソフトウェアの著作権所有者は，流通しているソフトウェアが自分の著作物，または，著作物の一部を流用したものであるかどうかを判定できる．これにより，不正者によるソフトウェアの著作権侵害を間接的に防ぐことができる．

### 2.1 外部からの観測，改ざんを難しくする手法

この節では，外部からの観測，改ざんを利用したソフトウェアの著作権保護手法の代表として，ソフトウェアの暗号化，サーバサイド実行，およびソフトウェア難読化について説明する．

**■ ソフトウェアの暗号化**

ソフトウェアを暗号化することによってプログラムの解読を防止することができる [4]．しかし，暗号化されたソフトウェアを実行するには，いったん機械語命令に復号し，実行する必要がある．もしくは，復号機能をもつ特殊な実行環境で実行する必要がある．

**■ サーバサイド実行**

サーバサイド実行では，ソフトウェアを private part と public part に分割する．ここで，private part には，ソフトウェア内の秘密のデータや重要なアルゴリズムが含まれ，public part には，利用者に公開しても差し支えないデータやアルゴリズムしか含まれないように分割を行う．private part は著作権所有者のサーバに保管される．一方，public part には，そのサーバに対するクライアント機能を持たせて，各利用者に配布する．このソフトウェアの利用する場合，ユーザからの入力に応じて，public part が private part に処理を要求する．private part はこの要求に対し，処理を行い，結果を public part に送る．サーバサイド方式では，private part が利用者の手元に渡ることはないので，private part の安全性は完全に保証される．

しかし，ソフトウェアの利用時に private part と，public part の間の通信が必要となる．さらに，多くの利用者がこの技術で実現されたサービスを同時に利用した場合，サーバの負担は大きくなる．

**■ ソフトウェア難読化**

ソフトウェア難読化（以後，単に難読化と呼ぶ）とは，与えられたプログラムを機能を保ちつつ，解読が難しいプログラムに変換することである．難読化されたプログラムは一般のプログラムと同様に，計算機上でそのまま実行することが可能である．プログラムを難読化してから利用者に配布することによって，利用者や第三者によってプログラムが解読されてしまう危険性を軽減できる．

これまでにも数多くの難読化手法が提案されている．門田らは，ループを含む C のプログラムを自動的に難読化する手法を提案し [8]，Collberg らはダミーコードの挿入，データ構造の複雑化，制御構造の複雑化による Java の難読化手法を提案した [2]．安全性の理論的根拠を持つ難読化手法も提案されている．Wang ら [3] は，分岐アドレスの決定問題が NP-Hard であることを示した．さらに，配列とポインタを利用した間接参照による C の手続き内解析を困難にする難読化手法を提案した．小木曽らは，ポインタのアドレスの決定問

題がNP-Hardであることを示し，関数ポインタを用い，関数呼び出しを複雑にする手法を提案している[9]．この手法は，関数ポインタを利用しており，control flow graph, call graphを共に静的に決定できないようにする．このため，手続き内のcontrol flow graphのみを静的に決定できないようにするWangらの手法と比較して，静的解析はより困難になる．また，刑部らは，メソッドオーバロードが存在し，かつインタフェースを実装したクラスが存在するとき，手続き間の正確なpoint-toを求める問題はNP困難であることを示し，インタフェースおよびメソッドオーバロードの導入による難読化手法を提案した．この難読化手法は，難読化手法[3, 9]と同様な安全性を持つ．

セキュリティ技術によってコンテンツの保護を行う際には，コンテンツの価値に対して必要かつ十分な安全性を実現することが理想である．難読化手法を用いてソフトウェアを保護する際にも，ソフトウェアの価値に応じた安全性を実現しなければならない．この安全性を証明するためには，難読化の効果を定量的に評価できることが必要である．しかし，これらの難読化手法の安全性に対する定量的な評価は十分に与えられていないのが現状である．

## 2.2 ソフトウェアの同一性を証明する方法

ソフトウェアの同一性を証明する手法には，電子透かしとプログラム指紋がある．

### ■ 電子透かし

従来，画像データ，音声データ，テキスト文書等の著作物に電子透かしを挿入する方法が盛んに研究されてきた．これらのコンテンツは冗長度が高く，比較的容易に多くの情報量を埋め込むことができる．一般に電子透かしは利用者の攻撃によって容易に消されないことが重要である．画像データや音声データ中の電子透かしも容易には除去することは不可能である．多くの場合，透かしを除去する過程において，画像そのものを劣化させてしまうためである．

一方，プログラム言語は画像，音声，一般的なテキスト文章と比較して，冗長度が極めて低い．そのため，効率よく透かし情報を埋め込むことは難しい．透かしを埋め込むための手法としては，可能な範囲で命令の順番を入れ替えたり，冗長な命令を挿入したりすることが考えられる．ところが，このようなプログラムの冗長性を利用する方法は，透かしを挿入する方法を利用者に知られると，透かしを容易に消される恐れがある．しかし，最近になって，プログラム言語に対する効率的な透かし埋め込み手法が提案されつつある．Javaを対象とする透かしの提案は[11, 12]である．

**(1) 著作権所有者の識別情報を埋め込む場合**

ソフトウェアの著作権所有者の識別情報を電子透かしとして埋め込む手法は，大きく2つの用法がある．

1つは紙幣に代表されるようにソフトウェアの正当性を示すために用いられる．我々は紙幣の透かしを確認して，それが日本銀行によって発行された正当な紙幣であることを確認できる．この場合，透かしは全ての人にとって認識可能なものでなければならない．

もう1つは，ソフトウェアの著作権所有者がそのソフトウェアの所有権を主張するのに用いられる．あるソフトウェアの製作者がソフトウェア内に自らの識別情報を透かしとして埋め込んでおいたとする．このとき，他人がこのソフトウェアを入手して，不正に著作権を主張しようとした場合，著作権所有者はあらかじめ埋め込んでおいた自らの識別情報を復元することによって，コンテンツの盗用を容易に立証できる．この場合透かしは著作権所有者以外の人間は識別できる必要性はない．むしろ，他人が識別できたとすると，透かしに対して改ざん・除去による攻撃が行われる可能性がある．この場合の電子透かしにはこれらの攻撃に対する耐性が要求される．

北川ら[11]は，JAVAのプログラムに対して任意の数値列を透かしとして埋め込む手法を提案している．この手法では透かしを格納するための変数を用意し，その変数に透かし情報を表す数列を格納する．透かしの取り出しには特定のクラスファイルを透かし取り出し用のクラスファイルに変換する．しかしこの手法では透かしを検出しようとする際に取り出し用のクラスファイルに置き換えるべきクラスファイルが存在しない場合がある．このときは検出不可能になる．

門田ら[12]は，JAVAのプログラムの数値オペランド部分，オペコード部分に透かしを埋め込む手法を提案している．この手法ではソースコードに対してダミーのメソッドを追加することによって

透かしを埋め込むための領域を確保する．コンパイル後のクラスファイル中の数値オペランド，オペコード部分に透かし情報を埋め込む．この手法では一つのクラスファイルが盗用された場合でも透かし挿入部分を調べることにより簡単に透かしが復元できる．この手法に基づいたツール [1] も公開されている．

**(2) 利用者の識別情報を埋め込む場合**

利用者の識別情報を電子透かしとして埋め込む手法は，主に，ソフトウェアの不正コピーや再配布を検知するために用いられる．例えば，著作権所有者がソフトウェアを利用者に配布する際に，利用者の識別情報を，何らかの透かしとしてソフトウェア内に埋め込んでいたとする．配布されるソフトウェアの透かし部分は各利用者ごとに異なっている．このとき，ある利用者がソフトウェアをコピーして不正に再配布したとする．利用者は不正に出回っているコピーを入手し，電子透かしから利用者の識別情報を復元する．これにより，不正なコピーや再配布を行ったという事実を立証できる．

埋め込まれた利用者の識別情報は，不正なコピー・再配布を行った人物を特定するための情報であるので，この情報の改ざんは，前節で述べた開発社の識別情報の改ざんに比べて，深刻なものになると考えられる．

しかし，利用者の識別情報を透かしとして埋め込む場合，各利用者のソフトウェアの透かし部分は異なっている．このため，複数の利用者が結託すれば，透かしが埋め込まれた場所が特定される可能性がある．ソフトウェアに埋め込まれる透かしデータは一般的に実行には必要とされない冗長なものであるため，容易に除去，あるいは，改ざんを行うことができる．

■ **プログラム指紋**

プログラム指紋では，あらかじめ情報の埋め込みを行わずに，ソフトウェアの識別を行う方法である [13, 14]．ソフトウェア P の著作権所有者は，何らかのプログラム指紋抽出法 B によって，ソフトウェア P の固有の特徴の集合 B(P) を抽出しておく．この B(P) がソフトウェア P のプログラム指紋となる．不正者がソフトウェア P を盗用して再配布した場合，再配布されたプログラムからは，プログラム指紋 B(P) が得られるため，著作権所有者は盗用の事実を立証できる．

また，不正者はプログラム指紋の改ざんを目的として，ソフトウェアの配布前に等価変換による攻撃を行う可能性がある．ここで，等価変換とは，プログラムの機能を保った変換であり，難読化，最適化，リファクタリングなどがこれに当てはまる．ここでは，前述の難読化がプログラム指紋に対する攻撃方法となりうる．この場合にも，プログラム抽出手法が等価変換による攻撃に耐性を持っていれば，変換後のプログラム P' からも B(P) を抽出でき，盗用の事実を立証できる．また，正当な著作権所有者が開発した別のプログラム Q からは，B(P) とは異なるプログラム指紋 B(Q) が得られる必要がある．

これらの機能を実現するために，プログラム指紋は以下の 2 つの条件を満たすことが必要である．

**条件 1** プログラム $P$ と，プログラム $P$ と機能が異なるプログラム $Q$ に対し，$B(P) \neq B(Q)$ となる．

**条件 2** プログラム $P$ と，プログラム $P$ を等価変換されることによって得られたプログラム $P'$ に対し，$B(P)=B(P')$ となる．

# 3 カプセル化構造の破壊による Java ソースコードの難読化手法

## 3.1 考え方

提案する難読化手法では，Java のオブジェクト指向言語としての特性に着目する．Java では，データ構造（フィールド）とそれに対する操作（メソッド）をクラスとしてひとまとめに定義し，クラス内の細かい仕様や構造を外部から隠蔽している．このことをカプセル化という．クラス外部からは，公開されたメソッドを利用することでしかフィールドを操作できない．この結果，各クラスの独立性は高い．一方，Java における静的フィールドや静的メソッドは，特定のオブジェクトではなく，クラスそのものに属するフィールドやメソッドである．このため，静的フィールドに対して，任意のクラスから“クラス名.フィールド名”の形式で参照，代入を行うことが可能である．同様に静的メソッドは，“クラス名.メソッド名”の形式で任意のクラスから呼び出すことが可能である．そこで，提案手

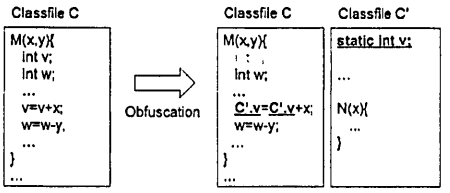


図 1: メソッド内の局所変数の移動による難読化の例

法では，クラス内の任意のメソッドの局所変数と命令群を他のクラスに移動する．このことにより，プログラムにおけるクラス間の呼び出し関係が複雑になり，プログラムの解読が難しくなる．以降の節では，局所変数を移動する方法と，命令群を移動する方法について述べる．

### 3.2 メソッド内の局所変数の移動方法

任意のクラスのメソッドに属する局所変数を移動する手順を説明する．

(1) **移動する局所変数の選択**
メソッド内の局所変数の中から難読化の対象となる局所変数を任意に選ぶ．

(2) **静的フィールドとしての再定義**
難読化の対象となる局所変数を配置するクラスを任意に定め，このクラス内に静的フィールドとして定義する．

(3) **修正**
メソッド内で難読化の対象となる局所変数に対して，参照および代入を行っている箇所について修正をする．つまり，“クラス名.フィールド名”の形式で (2) で定義した静的フィールドに対して参照および代入を行う．

(4) **削除**
難読化の対象となる局所変数の変数宣言を削除する．

局所変数の移動による難読化の例を図 1 に示す．はじめに，クラス C 内で定義されたメソッド M の局所変数の中から，難読化の対象となる局所変数 v 選ぶ．次に，難読化の対象となる局所変数 v を配置するクラス C' を選ぶ．クラス C' 内で静的フィールドとして，変数 v を新たに定義する．さらに，クラス C のメソッド M 内で局所変数 v に対して，参照および代入を行っている箇所について，クラス C' の静的フィールドである変数 v に対して，それぞれ C'.v の形式で参照および代入を行うように修正する．最後に，難読化の対象となる局所変数 v の宣言を削除する．

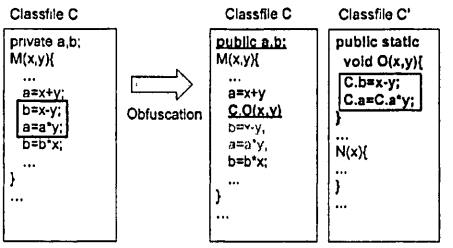


図 2: メソッド内の命令群の移動による難読化の例

### 3.3 メソッド内の命令群の移動方法

任意のクラスのメソッド内の命令群を他のクラスに移動する手順を述べる．

(1) **移動する命令群の選択**
メソッド内の全命令からプリミティブ型の局所変数への代入命令を除いたものの中から，難読化の対象となる任意の命令群を選ぶ．

(2) **静的メソッドへとしての再定義**
難読化の対象となる命令群を配置するクラスを任意に選び，このクラス内に静的メソッドとして定義する．元のメソッド内において，難読化の対象となる命令群が参照していた局所変数については，引数として呼び出し時に与えられるようにする．

(3) **メソッド呼び出しの追加**
(1) で難読化の対象なる命令群の直前に，(2) で定義したメソッドへの呼び出しを記述する．形式は“クラス名.変数名 (引数)”となる．

(4) **修正**
(2) で定義したメソッドにより参照および代入が行われているフィールドの修飾子は **public** に変更する．このことにより，他のクラスに配置したメソッドから，当該フィールドの参照および代入が可能になる．

(5) **削除**
難読化の対象となる命令群をメソッド内から削除する．

命令群の移動による難読化の例を図 2 に示す．はじめに，クラス C 内で定義されたメソッド M の内部にあ

る連続した命令 b=x-y と a=a*y の2つの命令を難読化の対象となる命令群として選ぶ．次に，難読化の対象となる命令群を配置するクラス C' を任意に選ぶ．クラス C' 内で静的メソッド O を新たに定義する．この静的メソッドは難読化の対象となる命令群で構成される．クラス C のメソッド M の局所変数である x，y の値は，メソッド O の引数として呼び出し時に与える．さらに，クラス C のメソッド M 内で難読化の対象となる命令群の直前の箇所に，クラス C' 内に新たに定義したメソッド O への呼び出しを記述する．メソッドの呼び出し形式は C'.O(x,y) となる．最後に，クラス C 内で難読化の対象となる命令群を削除する．

Java ではプリミティブ型の引数は値渡しで，メソッドに渡される [5]．このため，局所変数への代入命令を静的メソッドとして他のクラスに移動しても，静的メソッドからは局所変数の値を変更することは不可能である．これが，手順 (1) においてプリミティブ型の局所変数への代入命令を除外した理由である．なお，オブジェクト型の引数は参照渡しでメソッドに渡される．この場合は静的メソッドから局所変数への代入操作を行うことが可能であり，オブジェクト型の局所変数への代入命令は静的メソッドとして他のクラスに移動することが可能である．また，上記で説明した局所変数の移動による難読化手法を適用すれば，局所変数を他のクラスの静的フィールドに変換することができ，この問題を解決することが可能である．

### 3.4 考察

門田らは，難読化手法 $\tau$ が満たすべき2つの条件と満たすことが望ましい1つ条件を挙げている [8]．

**条件 1 出力の等価性**
難読化前のプログラム $P$ と難読化後のプログラム $\tau(P)$ は同じ出力を返す．

**条件 2 プログラムの解読時間**
難読化後のプログラム $\tau(P)$ の解読は，難読化前のプログラム $P$ の解読よりも時間がかかる．

これらの2つの条件は，難読化手法 $\tau$ が満たすべき条件である．

条件 3 プログラムの実行時間
難読化後のプログラム $\tau(P)$ の実行時間と難読化前のプログラム $P$ の実行時間はほぼ変わらない．

この条件3は，難読化手法 $\tau$ が満たすことが望ましい条件である．

提案した難読化手法がこれらの条件を満たしていることを以降で述べる．

#### 3.4.1 出力の等価性

あるメソッドの局所変数を静的フィールドとして他のクラスに移動させても，クラス名，フィールド名を指定できれば，任意のメソッドから参照や代入を行うことは可能である．このため，局所変数の移動による難読化手法を適用しても出力の等価性は保たれる．また，命令群を静的メソッドとして他のクラス移動した場合の変数のスコープの違いは，静的メソッド側から見えない変数を引数によって渡すことで解決することができる．このため，命令群の移動による難読化手法を適用しても出力の等価性は保たれる．

#### 3.4.2 プログラムの解読時間

一般の Java のプログラムでは，フィールドとメソッドがひとまとめに定義されている．カプセル化によって，クラス外からは，公開されたメソッドを利用することでしかフィールドを操作できない．このため，各クラスの独立性は高く，各クラスファイルを個別に解読できれば，プログラム全体の解読も容易である．一方で，提案手法により難読化されたプログラムはカプセル化構造が破壊されている．この状況では，各クラスの独立性は弱まっている．このため，個別のクラスファイルの解読が十分な意味をなさなくなる．すなわち，このプログラムを構成しているクラスファイルを解読するためには，そのクラス内のメソッドによって，参照および代入が行われている静的フィールド，静的メソッドが属するクラスを調べる必要があり，解読に必要な時間は，難読化前のソースコードと比較して長くなる．

次に，難読化の耐性について考察する．ダミーの命令の挿入や制御構造の複雑化により難読化されたソースコードをコンパイルした場合，コンパイラの最適化機能によりダミーの命令部分が除去され，制御構造が単純化されたクラスファイルが得られる可能性がある．このクラスファイルを逆コンパイルすることによって，難読化前のソースコードに近いソースコードを復元することができる．しかし，一般的なコンパイラ (javac) では，コードの最適化を行うことは可能であるが，クラス構造の最適化を行うことができないので，本手法による難読化はコンパイル，逆コンパイルによる攻撃に対して，耐性があるといえる．

また，攻撃者が難読化されたクラスファイルを解読する際には，そのクラスからの他クラスのフィールド，メソッドに対して参照および代入を行っている箇所と，該当するクラスでこれらが定義されている箇所の関連を調べる．このとき，意味を持つフィールド名やメソッド名が残っていれば，攻撃者は容易に関連を調べることができる．難読化を行った後，各クラスのフィールド名，メソッド名を意味がないものに変更することによって，この問題を解決することができる．

#### 3.4.3 プログラムの実行時間

局所変数を静的フィールドに変換しても，変数に対する参照および代入を行う際のバイトコード命令の1つが置き換わるだけである．そこで，実際にJavaプログラムを作成し，局所変数と静的フィールドに対する参照および代入にかかる時間を測定した．この結果，局所変数を静的フィールドに変換した場合の実行時間の増加は，平均8%程度であった．また，1回のメソッド呼び出しのオーバヘッドは小さいと推測される．さらに，命令群を他のクラスに移動しても，制御構造の複雑化や意味のない命令の追加などを行わない限り，その命令群の処理は同じである．このため，命令群自体の実行時間はほぼ変わらないと考えられる．ただし，難読化を繰り返し適用すると，ソースコードの行数が増加する．このため，実行時間が増加すると考えられる．

## 4 難読化の評価

### 4.1 評価尺度

以下に，3章で提案したJavaソースコードに対する難読化手法に対する評価尺度を与える．この評価尺度は複数のクラスにまたがるメソッド，フィールドの呼び出し回数に基づいた尺度である．最初に，クラスCの複雑度$e(\mathrm{C})$を他のクラスで定義されているフィールド，メソッドのうちクラスCから参照および代入が行われているものの総数によって定義する．次に，プログラムPの複雑度$E(\mathrm{P})$を，そのプログラム内で定義されているすべてのクラスの複雑度の総和によって定義する．すなわち，

$$E(\mathrm{P}) = \sum_{\mathrm{C} \in \mathrm{P}} e(\mathrm{C})$$

となる．最後に難読化$\tau$の効果$\mathit{Effect}(\tau)$を難読化後のプログラム$\tau(\mathrm{P})$の複雑度と難読化前のプログラムPの複雑度の差によって定義する．すなわち，

$$\mathit{Effect}(\tau) = E(\tau(\mathrm{P})) - E(\mathrm{P})$$

となる．

この評価尺度におけるプログラムの複雑度は，定義によりプログラムで定義されているクラス数に伴い増加する．また，あるクラスファイルに対して，メソッド内の局所変数や命令群を他のクラスに移動すると，クラス間にフィールドおよびメソッドの呼び出し関係が発生する．したがって，このクラスの複雑度は増大する．プログラム内で定義されているクラス数が多く，クラス間にまたがる呼び出し関係が多いほどプログラムの解読は難しくなる．このため，評価尺度によるプログラムの複雑度は妥当なものであると考えられる．また，評価尺度による難読化の効果は，難読化後のプログラムと難読化前のプログラムの差によって定義されている．難読化後のプログラムの複雑度が大きくなれば，難読化の効果も大きくなるため，評価尺度による難読化の効果も妥当なものであると考えられる．

### 4.2 実験

#### 4.2.1 実験手順

提案した評価尺度を用いて，難読化手法を評価するために，実際のプログラムにおける解読時間と難読化の効果の相関を調べる．実験は以下の手順に従って行う．

(1) **テストプログラムの作成**
今回の実験では，フィボナッチ数列の最初の20項を求めるプログラムP0のソースコードを作成した．

(2) **難読化の適用**
プログラムのソースコードに3章の難読化手法を手作業で適用する．今回の実験では提案した難読化手法を適用して，5つの難読化されたプログラムP1, P2, P3, P4, P5のソースコードを作成した．各ソースコードについての詳細は表1にまとめる．5つの難読化されたプログラムは，プログラムP0と同様にフィボナッチ数列の最初の20項を出力する．ただし，各ソースコードの複雑度および施された難読化の効果は異なっている．

(3) **被験者による解読時間の測定**
難読化前のソースコードと難読化の効果が異なる6つのプログラム(P0, P1, P2, P3, P4, P5)を

表 1: 各ソースコードの特性

| Program | P0 | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|---|
| Size (B) | 369 | 563 | 924 | 959 | 1428 | 1939 |
| Lines | 26 | 39 | 48 | 54 | 78 | 105 |
| Methods | 2 | 4 | 4 | 6 | 10 | 19 |
| Classes | 2 | 3 | 4 | 5 | 5 | 6 |
| Complexity | 2 | 4 | 19 | 29 | 34 | 45 |
| *Effect* | 0 | 2 | 17 | 27 | 32 | 43 |
| Execution time ($10^{-6}$ s) | 539 | 664 | 673 | 747 | 817 | 831 |

5 人の被験者に渡す．被験者は全員大学院生である．被験者は各プログラムのソースコードの解読を行う．ただし，“慣れ”が生じないようにするため，ソースコードを見る前に解読する順番をランダムに決めておくように指示した．また，ソースコードの解読の間には半日以上の時間をあけてもらった．プログラムの開始から計算結果が出力されるまでの実行パス，実行パスに沿った演算，およびデータの流れを理解できた時点で解読終了とみなす．最後に被験者は解読時間を 10 秒単位で記録する．

#### 4.2.2 実験結果

表 1 より，難読化したプログラム P5 の実行時間は難読化前のプログラム P0 に比べて，54%程度するが，ソースコードのサイズの増加率（425%）や行数の増加率（303%）と比べるとその増加の割合は小さかった．図 3 に各被験者による解読時間と難読化の効果との関係を示す．被験者毎の個人差は若干見られるものの，解読時間は難読化の効果に伴い増加していた．次に，図 4 に 5 人の被験者の平均解読時間と難読化の効果との関係を示す．ソースコードの解読時間は，難読化の効果よりも速く増加することが示された．

### 4.3 考察

難読化された全てのプログラム (P1,P2,P3,P4,P5) は，難読化前のプログラム P0 と同様に，フィボナッチ数列の最初の 20 項を出力した．このことにより，難読化したプログラムの出力の等価性を確認した．次に，難読化手法を繰り返し適用すると，難読化の効果も増加することを確認した．それに伴い，効果に対するその増加割合は，効果が増大するほど大きくなる傾向がある．この結果，提案手法はプログラムの解読時間に関する難読化の条件を満たしていることを確認できる．

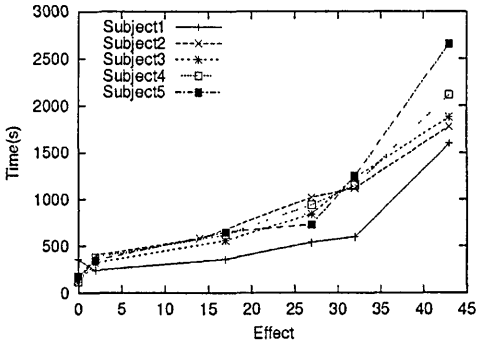


図 3: 5 人の被験者の解読時間と難読化の効果との相関

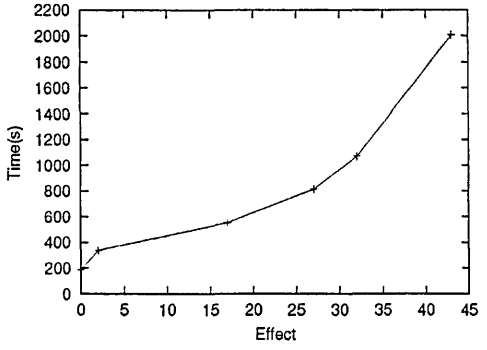


図 4: 5 人の被験者の平均解読時間と難読化の効果との相関

また，難読化後のプログラムの実行時間は難読化前のプログラムの実行時間とほぼ変わらないことが望ましいが，提案手法では，難読化の効果に伴い実行時間は増加する．しかし，難読化の適用により解読時間が 10 倍以上になる場合（P5 の場合）にも，実行時間の増加は 54%程度にとどまることを示した．

## 5 まとめ

本論文では，カプセル化構造による Java ソースコードの難読化手法を提案した．この手法では，任意のメソッド内の局所変数および命令群を他のクラスに移動する．このため，各クラスの独立性が弱まり，個別のクラスファイルの解読が十分な意味をなさなくなる．この手法により難読化されたプログラムを構成するクラスファイルを解読するためには，そのクラス内のメソッドにより，参照および代入が行われれいる静的フィールド，静的メソッドが属するクラスも解析する必要があり，プログラムの解読は困難となる．

さらに，提案した難読化手法に対する効果の評価尺度の提案を行った．この評価尺度は，複数のクラス間にまたがるメソッド，フィールド参照および代入の回数に基づき定義されたものである．最後に，この評価尺度を利用して，提案手法の評価実験を行った．この結果，提案手法を用いて難読化を行った場合のプログラムの出力の等価性と解読時間の増加を確認した．また，難読化手法の適用により解読時間が10倍以上になる場合にも，実行時間の増加は54%程度にとどまることを示した．

今後の課題としては，大規模なプログラムでの提案手法の評価がある．

## 参考文献


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

[2] C.Collberg, C.Thomborson and D.Low, "A taxonomy of obfuscating transformations," Technical Report of Dept. of Computer Science, U. of Auckland, No.148, New Zealand, 1997.

[3] C. Wang, J. Hill, J. Knight and J. Davidson, "Software tamper resistance: obfuscating staticanalysis of programs," Technical Report SC-2000-12, Department of Computer Science, University of Virginia, 2000.

[4] D.J. Albert and S.P. Morse, "Combating software piracy by encryption and key management," IEEE J. Computers, pp.68-73, April 1984.

[5] J. Gosling, B. Joy, G. Steele and G. Bracha, The Java Language Specification Second Edition, Addison Wesley Professional, Pearson Education Company, 2000.

[6] T.Lindholm and F.Yellin,The Java Virtual Machine Specification, Addison Wesley Longman, Pearson Education Company, 1999.

[7] 刑部裕介，双紙正和，宮地充子，"オブジェクト指向言語の難読化の提案，"信学技報（ISEC02-6），pp.33–38, May 2002.

[8] 門田暁人，高田義広，鳥居宏次，"ループを含むプログラムを難読化する方法の提案，"信学論（D-I），vol.J80-D-I, no.7, pp.644–652, July 1997.

[9] 小木曾俊夫，刑部裕介，双紙正和，宮地充子，"手続き間呼び出しの関係に着目した難読化手法の提案とその評価，"2002年暗号と情報セキュリティシンポジウム（SCIS2002），分冊1，pp.353–358, Jan. 2002.

[10] 内藤篤，"ソフトウェア著作権関連判決集，"
http://www.venture.tao.go.jp/copyright/

[11] 北川隆，椙勇一，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，"1998年暗号と情報セキュリティシンポジウム（SCIS'98），pp.231–236, 1998.

[12] 門田暁人，飯田元，松本健一，鳥居宏次，一杉裕志，"プログラムに電子透かしを挿入する一手法，"1998年暗号と情報セキュリティシンポジウム（SCIS'98），pp.243–248.1998.

[13] 玉田春昭，神崎雄一郎，中村匡秀，門田暁人，松本健一，"Javaクラスファイルからプログラム指紋を抽出する方法，"信学技報，Vol. 103, No.95, pp.127–133, 2003.

[14] 福島和英，田端利宏，櫻井幸一，"Javaクラスファイルの等価変換に耐性を持つプログラム指紋抽出法，"情処研報，Vol. 2003, No.126, pp.81–86, 2003.