NAIST-IS-DD0361018

博士論文

バースマークと動的名前解決による
ソフトウェアプロテクション

玉田 春昭

2006年2月2日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である。

玉田 春昭

審査委員：

松本 健一 教授	（主指導教員）
関 浩之 教授	（指導教員）
楫 勇一 助教授	（指導教員）
門田 暁人 助教授	（指導教員）

バースマークと動的名前解決による ソフトウェアプロテクション*

玉田 春昭

内容梗概

本論文では，ソフトウェアの盗用 (他人の作成したソフトウェアを許可なく自分のソフトウェアに組み込む行為)，および，ソフトウェアクラック (ソフトウェア内部に含まれる秘密情報の解析・改ざんなどの行為) をそれぞれ抑止するための 2 つの方法を提案する．

まず盗用の抑止を目的として，ソフトウェアバースマークという概念を提案する．バースマークとは，各ソフトウェア固有の特徴を表す値の集合であり，ソフトウェアが改ざんされてもほとんど変化しないという "保存性" と，異なるソフトウェアからは全く異なるバースマークが抽出される "弁別性" の 2 つの性質を持ことが望まれる．本論文では，Java 言語を対象に変数初期値バースマーク, メソッド呼び出し系列バースマーク, 継承関係バースマーク, 使用クラスバースマークの 4 種類のバースマークを提案し，保存性，弁別性の評価を行った．まず保存性の評価では，Apache Ant に 4 種類の難読化ツールを適用し，難読化前後のクラスファイルからバースマークを抽出し，比較した．その結果，バースマークの類似度の平均は 0.940 であり保存性があることを示した．また，弁別性評価では，Java バイトコードを扱う 3 つのライブラリ BCEL, javassist, bloat に含まれるクラスファイルから 4 種類のバースマークを抽出した結果，0.962 の割合で異なるクラスファイルを弁別できることを確認した．

---

* 奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 博士論文, NAIST-IS-DD0361018, 2006 年 2 月 2 日.

次に，クラックの抑止を目的として，動的名前解決を利用して，ソフトウェアの名前難読化を行う方法を提案する．名前難読化とはプログラム中に現れる名前(識別子) を別のものに付け替える難読化である．しかし，従来法ではユーザが定義した名前しか隠すことができず，標準ライブラリのようなシステムが提供するライブラリに含まれるメソッドやクラス名を隠すことはできない．提案方法は，メタプログラミングに用いられる動的名前解決機構を難読化に導入することで，ユーザライブラリの呼び出しのみならず，従来隠すことのできなかった標準ライブラリの呼び出し (クラス参照やメソッド呼び出し，フィールドの参照・代入のことを言う) を隠すことが可能となる．これにより，秘密にすべき呼び出し (例えば認証処理) を隠すことができ，ソフトウェアをクラックから守ることができる．実用規模のライブラリである Jakarta Commons Digester に提案手法を適用した結果，2,359 のフィールド参照，673 のフィールドへの代入，197 のオブジェクト生成，7,351 のメソッド呼び出しの全てのクラス名，メソッド名，フィールド名を隠蔽することができた．

キーワード

ソフトウェア保護，ソフトウェアバースマーク，電子透かし，ソフトウェア難読化，リフレクション

Software Protection by Birthmark and Dynamic Name Resolution*

Haruaki Tamada

Abstract

This dissertation proposes two methods which prevent software theft and software cracking. First, to detect the theft of Java class files efficiently, I propose a concept of *software birthmarks*, which are unique and native characteristics of each software implementation. Ideally, the birthmarks should satisfy the following properties: (a) *preservation* – the birthmarks should be preserved even if the original class file is tampered with, and (b) *distinction* – independent class files must be distinguished by completely different birthmarks. Taking (a) and (b) into account, I propose four types of birthmarks for Java class files. To show the effectiveness of the proposed birthmarks, I conducted two experiments. The first experiment showed that the proposed birthmarks are sufficiently robust against automatic program transformation (the average of similarity of birthmarks was 0.940). The second experiment showed that the proposed birthmarks successfully distinguished non-copied files in the following Java bytecode engineering libraries: BCEL, javassist, and bloat (0.962 of given class files were distinguished).

Next, this dissertation proposes a name obfuscation method based on *dynamic name resolution* to prevent software cracking. While conventional name obfuscation cannot hide symbol names included in system libraries, the proposed obfuscation method hides system library calls (i.e. method invocation, field reference/assignment and object instantiation) by dynamic name resolution. Using

* Doctoral Dissertation, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0361018, February 2, 2006.

the proposed method, names appeared in system library calls are statically encrypted. Then, the names are decrypted and dynamically resolved at runtime. A case study showed that the proposed method could obfuscate the all names of 2,359 field references, 673 field assignments, 197 object instantiation, and 7,351 method invocations included in Jakarta Commons Digester.

Keywords:

Software Protection, Software Birthmark, Software Watermark, Software Obfuscation, Reflection

関連発表論文

第2章に関連する発表論文

学術論文誌

1. Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto, Java birthmarks —detecting the software theft—, *IEICE Transactions on Information and Systems*, Vol.E88-D, No.9, pp.2148–2158, September 2005.

国際会議

1. Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto, 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, Feburary 2004. Innsbruck, Austria.

研究会・シンポジウム

1. 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の提案, 電子情報通信学会 技術研究報告 情報セキュリティ研究会, ISEC2003-29, pp.127–133, July 2003.

テクニカルレポート

1. Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto, Detecting the theft of programs using birthmarks, Information Science Technical Report NAIST-IS-TR2003014 ISSN 0919-9527, Graduate School of Information Science, Nara Institute of Science and Technology, November 2003.

第 3 章に関連する発表論文

特許出願

1. 玉田春昭, 門田暁人, 中村匡秀, 松本健一, プログラム変換装置，呼出し支援装置，それらの方法およびそれらのコンピュータ・プログラム, 特願 2005-171372, June 2005.

その他の発表論文

学術論文誌

1. 井垣宏, 中村匡秀, 玉田春昭, 松本健一, サービス指向アーキテクチャを用いたネットワーク家電連携サービスの開発, 情報処理学会論文誌, Vol.46, No.2, pp.314–326, February 2005.

国際会議

1. Takahiro Kimura, Haruaki Tamada, Hiroshi Igaki, Masahide Nakamura, and Ken-ichi Matsumoto, A visual framework for monitoring and controlling distributed service components, In *Proc. The first Korea/Japan Joint Workshop on Ubiquitous Computing & Networking Systems 2005 (UbiCNS 2005; IPSJ SIG Technical Reports 2005-UBI-8)*, Vol.2005, pp.245–250, June 2005. Jeju, Korea.

2. Masahide Nakamura, Hiroshi Igaki, Haruaki Tamada, and Ken-ichi Matsumoto, Implementing integrated services of networkeded home appliances using service oriented architecture, In *Proc. International Conference of Service Oriented Computing (ICSOC2004)*, pp.269–278, November 2004. New York, USA.

3. Haruaki Tamada, Keiji Okamoto, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto, Dynamic software birthmarks to detect the theft of windows applications, In *Proc. International Symposium on Future Software Technology 2004 (ISFST 2004)*, October 2004. Xi'an, China.

研究会・シンポジウム

1. 玉田春昭, 井垣宏, 引地一将, 門田暁人, 松本健一, プログラミング実習におけるグループ開発プロセスの分析, ソフトウェア工学の基礎 XI, 日本ソフト

ウェア科学会 (FOSE 2005), pp.123–128, 近代科学社, November 2005.

2. 浜田美奈子, 玉田春昭, 中道上, 武村泰宏, 大平雅雄, マイクバーカー, プログラミング実習時の学習者の感情に着目した自発性測定手法の検討, 電子情報通信学会 技術研究報告 教育工学研究会, ET2005-32, pp.29–34, September 2005.

3. 浜田美奈子, 玉田春昭, 中道上, 武村泰宏, マイクバーカー, プログラミング実習における自発性測定のための感情と自発性の関連分析, 日本教育情報学会 第21回年会論文集, pp.192–195, August 2005.

4. 玉田春昭, 門田暁人, 中村匡秀, 松本健一, Java プログラムの動的解析のためのトレーサ埋め込みツール, 第46回プログラミング・シンポジウム報告集, pp.51–62, January 2005.

5. 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, 第46回プログラミング・シンポジウム報告集, pp.41–50, January 2005.

6. 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提案, ソフトウェア工学の基礎XI, 日本ソフトウェア科学会 (FOSE 2004), pp.85–88, 近代科学社, November 2004.

7. 井垣宏, 玉田春昭, 中村匡秀, 松本健一, サービス指向アーキテクチャを用いたホームネットワークシステムの設計と評価尺度, 電子情報通信学会 技術研究報告 ネットワークシステム研究会, NS2004-316, pp.127–133, March 2004.

8. 串戸洋平, 石井健一, 山内寛己, 井垣宏, 玉田春昭, 中村匡秀, 松本健一, Webサービスアプリケーションのソフトウェアメトリクスに関する考察, 電子情報通信学会 技術研究報告 ネットワークシステム研究会, NS2004-316, pp.113–118, March 2004.

9. 石井健一, 串戸洋平, 山内寛己, 井垣宏, 玉田春昭, 中村匡秀, 松本健一, 異なる連携方式を用いた web サービスアプリケーションの開発および評価, 電子情報通信学会 技術研究報告 ネットワークシステム研究会, NS2004-316, pp.107–112, March 2004.

目次

第1章　はじめに **1**
1.　研究の背景と目的 1
2.　論文構成 4

第2章　ソフトウェアバースマーク **6**
1.　はじめに 6
2.　関連研究 8
3.　準備 10
3.1　バースマークの定義 10
3.2　バースマークの満たすべき性質 10
4.　バースマーク 11
4.1　提案バースマーク 11
4.2　バースマークの類似度 17
5.　評価実験 18
5.1　`jbirth` 18
5.2　保存性評価 20
5.3　弁別性評価 23
6.　議論 25
6.1　現実的な提案手法の使用方法 25
6.2　静的バースマークと動的バースマークの比較 29
6.3　手動改変に対する耐性 30
7.　まとめ 31

第 3 章　動的名前解決を用いたソフトウェア難読化 33
1. はじめに 33
2. 準備 35
2.1 ソフトウェア難読化 35
2.2 名前難読化 36
2.3 従来手法 37
3. 動的名前解決を用いた名前難読化 39
3.1 キーアイデア 39
3.2 動的名前解決 40
3.3 動的名前解決を用いた名前難読化 42
4. 実装 44
4.1 動的名前解決支援クラス DynamicCaller の導入 45
4.2 クラスファイルの書き換え 46
4.3 難読化例 50
5. 評価 51
5.1 性能評価 51
5.2 セキュリティ評価 55
5.3 他の名前難読化手法との比較 56
6. まとめ 57

第 4 章　終わりに 58

謝辞 60

参考文献 62

付録 71
A. バースマークの規模 71
A.1 ソースコード行数に対するバースマークの要素数の分布 . 71
A.2 バースマークの要素数ごとのクラス数の分布 75
B. 難読化例 78

B.1 Hello World 78
B.2 Fibonacci 79

図 目 次

1.1 開発の局面におけるソフトウェアの不正利用の典型例 2
1.2 エンドユーザサイドにおけるソフトウェアの不正利用の典型例 . . 3

2.1 Java プログラム例 (Apache Ant における echo タスク) 12
2.2 図 2.1 に示すプログラムの変数初期値バースマーク 13
2.3 図 2.1 に示すプログラムのメソッド呼び出し系列バースマーク . . 15
2.4 2.1 に示すプログラムの継承関係バースマーク 15
2.5 2.1 に示すプログラムの使用クラスバースマーク 16
2.6 `jbirth` のスクリーンショット 19
2.7 各バースマークの保存性評価 (各バースマーク毎) 22
2.8 各バースマークの保存性評価 (4 つのバースマークを同時に使用) . 23
2.9 各バースマークの弁別性評価 (各バースマーク毎) 27
2.10 異なるクラス間でのバースマークの類似度 (4 つのバースマークを同時に使用) 28

3.1 サンプルプログラム (画像ビューワ) 38
3.2 従来の名前難読化 39
3.3 動的名前解決を用いた難読化 (図 3.1 のコード) 43
3.4 バイトコード書き換えルール 48
3.5 スタック上の引数を配列に格納する手順 49
3.6 難読化後のサンプルプログラム 52
3.7 Digster の実行結果 54
3.8 従来法と提案法の適用範囲 57

A.1 Apache Ant から抽出したバースマークの要素数の分布 71

A.2 Jakarta BCEL から抽出したバースマークの要素数の分布 72
A.3 bloat から抽出したバースマークの要素数の分布 72
A.4 javassist から抽出したバースマークの要素数の分布 73
A.5 Apache Ant から抽出したバースマークの要素数の分布（0〜30） 73
A.6 Jakarta BCEL から抽出したバースマークの要素数の分布（0〜30） 74
A.7 bloat から抽出したバースマークの要素数の分布（0〜30） 74
A.8 javassist から抽出したバースマークの要素数の分布（0〜30） . . 75
A.9 Apache Ant から抽出したバースマークの要素数の頻度 76
A.10 Jakarta BCEL から抽出したバースマークの要素数の頻度 76
A.11 bloat から抽出したバースマークの要素数の頻度 77
A.12 javassist から抽出したバースマークの要素数の頻度 77
B.13 HelloWorld.java (オリジナル) 78
B.14 HelloWorld.java (難読化後) 78
B.15 Fibonacci.java (オリジナル) 79
B.16 Fibonacci.java (難読化後) 80

表 目 次

2.1 バースマーク抽出の実行時間 18
2.2 実験 1 保存性評価の結果 21
2.3 実験 2 弁別性評価の結果 26

3.1 DynamicCaller のメソッド 45
3.2 難読化に要した時間 51
3.3 難読化したプログラムの DynamicCaller 呼び出し回数 53
3.4 難読化したプログラムの実行時間とファイルサイズ 54

第1章　はじめに

1. 研究の背景と目的

今日，情報化社会の進展に伴い，ソフトウェアシステムが担う社会的役割はますます高まっている．その一方で，ソースコードの流出や盗用，システムの安全性に関わるソフトウェアの解析や改ざんといった様々な海賊行為が問題となってきており，ソフトウェアの開発と利用の両局面におけるソフトウェア保護の必要性が高まっている．

まず，開発の局面におけるソフトウェア保護の問題を考える．従来，開発時のソフトウェアの権利の取り扱いは，使用許諾書 (ライセンス) や契約に基づいて行われてきた．例えば，オープンソースのライセンスの一つである GPL (GNU General Public License)[19] では，ソースコードは受け取ったそのままの状態であるならば，再配布は許可される．また，ソースコードを改変した場合，改変箇所と改変者を明らかにして，元と同じ GPL で配布すればライセンス違反に問われることはない．

しかし，開発者にとっての制約はライセンスで定められているのみであり，ソフトウェアが物理的に保護されているわけではない．また，ライセンスに違反してソフトウェアを別の開発に密かに利用（盗用）したとしても，ライセンス違反の事実を他者が知ることは必ずしも容易でない．そのため，ソフトウェアをライセンスに違反して使用することに対する技術的・心理的障壁は高くないのが現状である．現実に，ライセンス違反の事例は数多く報告されている．例えば，Mac OS X エミュレータである Cherry OS が，実はオープンソースソフトウェアである Pear PC のライセンスに違反してコピーされていた事例 [50] や，エプソンコーワ社が提供する Linux ドライバに GPL で提供される gettext が含まれていなが

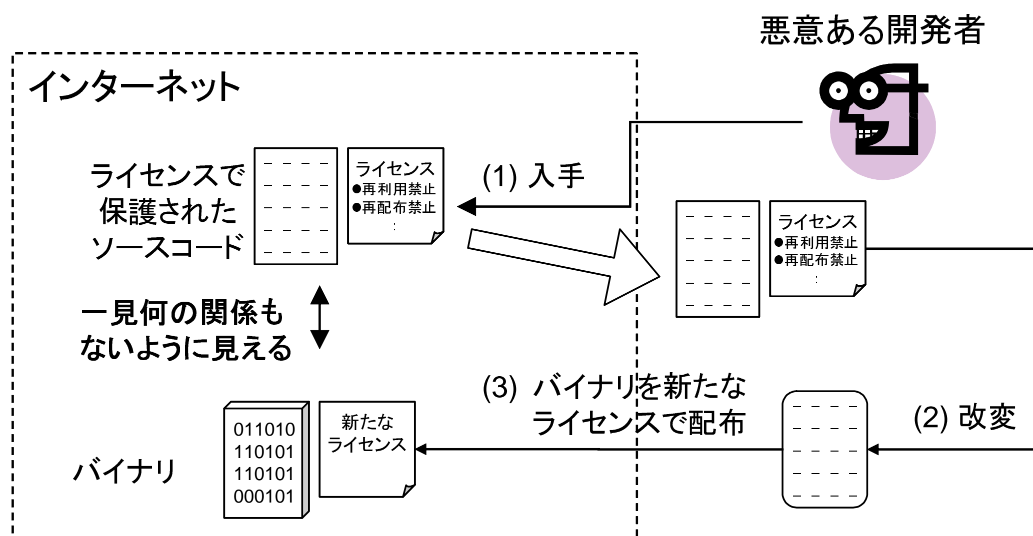

図 1.1 開発の局面におけるソフトウェアの不正利用の典型例

ら，ソースコードが提供されていなかった事例が挙げられる [52].

ライセンス違反の典型的なシナリオを図 1.1 に示す．まず，(1) 悪意ある開発者がインターネットを通じて配布されているソースコードを入手し，(2) ソースコードを改変し，あたかも新規に作成されたソフトウェアに見せかけて，(3) バイナリを新たなライセンスでインターネット上で配布する．

開発の局面におけるソフトウェア保護の難しさは，(1) で配布されたライセンス付きのソースコードと (3) で配布されたソフトウェアのバイナリが一見何の関係もないように見えることである．ソースコードが開示されていれば，コードクローン検出技術等 [25] を用いることで，ライセンス違反のコードが含まれていないかどうかを確認することは比較的容易に行えるが，バイナリプログラムのみが配布されている場合，ライセンス違反の有無を確認することは難しい．さらに言えば，盗用したソースコードを用いて開発が行われた場合，あたかも新規開発のソフトウェアとして（バイナリプログラムが）提供され，盗用の疑義を指摘することすら難しいことも多い．以上のことから，開発局面におけるソフトウェア保護の問題を技術的に解決するには，ソフトウェアのバイナリのみを用いて，盗用の事実を見つけ出すための技術が要求される．

一方，利用の局面においては，ソースコードが提供されることは少なく，また，

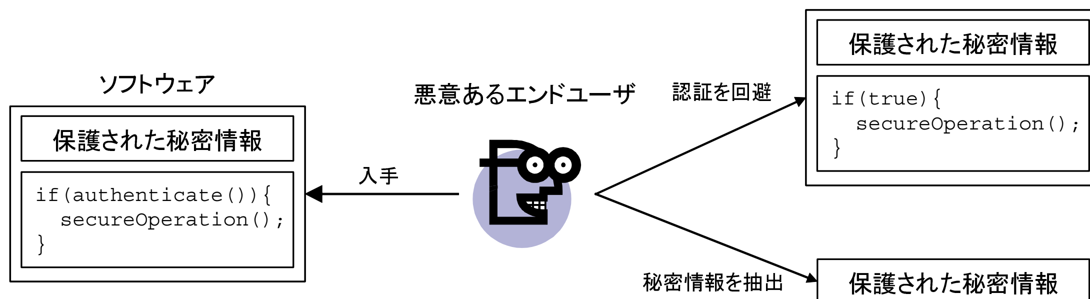

図 1.2 エンドユーザサイドにおけるソフトウェアの不正利用の典型例

エンドユーザの不正を阻止するため，様々な技術的保護機構が施されている場合が多い．広く一般に浸透した保護法方の一つにシリアル番号による保護が挙げられる．シリアル番号とは，ソフトウェアの各パッケージごとに割り当てられている固有の番号のことを指し，メーカ側での登録ユーザの管理に用いられる．また，ソフトウェアのインストール時にシリアル番号の入力をユーザに要求し，有効なシリアル番号でなければインストール作業を強制的に中断することで不正コピーの防止にも用いられる．

しかし，多くの場合，これらの保護機構もまたソフトウェアで実現されているため，悪意あるエンドユーザによるソフトウェア保護機構の破壊が，ソフトウェア利用局面における大きな問題となっている．典型的な例を図 1.2 に示す．図に示すように，悪意あるユーザは，ソフトウェア内部に含まれる秘密情報を抜き出したり，セキュリティのためのロジックを破壊，もしくはそのロジックを迂回するよう，ソフトウェアを改ざんすることが考えられる．具体的な事例としては，DVD プレイヤに含まれる DVD コンテンツを復号するための鍵が漏洩した事例 [44] や，iTunes Music Store で販売されている音楽コンテンツのコピープロテクトを無効化するパッチ [41] などの事例が報告されている．今日では，有線／無線放送システム，DVD プレイヤー，携帯電話等を始めとする，有料サービス関わるシステムにソフトウェアが用いられており，システムの安全性を確保する上で，エンドユーザからソフトウェアを保護することの重要性が著しく高まっている．

そこで，本論文ではソフトウェアの開発と利用の両局面からソフトウェアを保護するための手法を提案する．まず，開発の局面におけるソフトウェア保護のた

め，ソフトウェアバースマークの概念を提案する．ソフトウェアバースマークとはソフトウェア固有の特徴の集合のことであり，ソフトウェアそのものから直接抽出される．2 つのソフトウェアから抽出されたバースマークを比較することで，盗用の事実を発見できる．また，ソフトウェアバースマークの概念を満たす Java 言語を対象とした 4 種類のバースマークを提案する．それぞれ 変数初期値バースマーク, メソッド呼び出し系列バースマーク, 継承関係バースマーク, 使用クラスバースマークである．これらのバースマークはクラスファイル単位で抽出される．そのため，Java ソフトウェアに含まれる一部のクラスファイルが盗用されたとしても，その事実を検知することが可能である．

次に，利用の局面におけるソフトウェアの保護のため，不正な解析行為を防ぐことを考える．今日のソフトウェアは全てを 1 から作ることはなく，API と呼ばれる OS やミドルウェアなどのプラットフォームが提供する命令や関数の集合を利用して作成される．ソフトウェアは，プラットフォームが提供する様々な機能を API を呼び出すだけで利用することができる．

本論文では，秘密にすべき API 呼び出しを隠すためのソフトウェア難読化手法を提案する．難読化とは，与えられたプログラムをその振る舞いを変えないように理解しづらいプログラムへと等価変換する技術であり，これまでにも様々な手法が提案されてきている．提案手法は特にオブジェクト指向ソフトウェアを対象とした名前難読化 [62] の新たな手法である．提案手法は従来隠すことのできなかったライブラリ呼び出しを隠すことが可能である．これにより，ドングルへのアクセスや復号関数呼び出しなど秘密にすべき呼び出しを攻撃者から隠すことが可能になる．

2. 論文構成

本論文の主要部分は大きく 2 つの章から構成される．第 2 章ではバースマークについて，第 3 章では動的名前解決を用いたソフトウェア難読化について述べる．各章の構成は以下の通りである．

ソフトウェアバースマークの章では，第 1 節でソフトウェアバースマークの背

景について述べる．続く第 2 節でソフトウェアを区別する他の手段について述べ，ソフトウェアバースマークとの違いを明らかにする．そして，第 3 節でソフトウェアバースマークを定義し，ソフトウェアバースマークが満たすべき性質について述べる．そして，第 4 節で 4 種類の提案バースマーク，変数初期値バースマーク，メソッド呼び出し系列バースマーク，継承関係バースマーク， 使用クラスバースマークを定義する．続く第 5 節では提案バースマークがソフトウェアバースマークの性質を満たすことを，実験を通じて確かめる．そして，第 6 節では以下のことを議論する．まず第一に，ソフトウェアバースマークの典型的な使用法について述べる．提案バースマークはソフトウェアの静的解析によって抽出される．一方，ソフトウェアを実際に動作させ，実行中に得られる情報から抽出する動的バースマークも提案されている．この 2 つのソフトウェアバースマークの違いについて次に議論する．最後に，ソフトウェアバースマークの手動改変に対する耐性について議論する．第 7 節ではソフトウェアバースマークについてのまとめを行う．

続いて動的名前解決を用いたソフトウェア難読化の章では，第 1 節でこの研究の背景をより詳しく述べ，第 2 節でソフトウェア難読化の定義を行い，従来の難読化手法について述べる．そして，第 3 節で提案手法のキーアイデアを述べ，難読化を行うための手順について説明する．第 4 節では実際に提案手法を実現するための実装について述べ，難読化した例を示す．第 5 節では，提案難読化手法の評価を行う．一般に，リフレクションによる操作はオーバーヘッドが大きいことが知られている．そのため，第 5.1 節では，実用的なアプリケーションである Jakarta Digester を対象に，提案手法を適用したときの実行速度を計測する実験を実施する．そして，第 5.2 節では提案手法に対する考えられる攻撃を定性的に議論する．また，第 5.3 節では，従来の難読化手法と，提案手法を比較する．そして，最後に第 6 節で動的名前解決を用いたソフトウェア難読化についてのまとめを行う．

そして，本論文の最後の章である第 4 章で本論文全体のまとめと考察を行う．

第2章　ソフトウェアバースマーク

1．はじめに

今日，ソフトウェアの盗用が世界中で問題となっている．代表的な事例として，SCO グループと IBM の法廷闘争 [47]，GPL 違反 [52, 53, 50]，フリーウェアやシェアウェアの著作権侵害 [51] などが報告されている．BSA の報告によると，2003 年のソフトウェアのコピーや著作権侵害による被害額は 330 億ドルと推定されており [7]，ソフトウェアの盗用は，ソフトウェア産業に重大な被害をもたらしている．

しかし，ソフトウェアを盗用から守ることは容易ではない．今日では無数のソフトウェアが開発，配布されているため，その中から盗用の疑いのあるソフトウェアを発見することは難しい．さらに，盗用者がユーザインタフェースを変更したり，難読化ツール [13, 14] を用いた場合，ソフトウェアの盗用を発見および立証することは極めて困難となる．

一方，近年 Java 言語で作成されたアプリケーションが広く流通している．Java 言語では，クラスファイルの集合がアプリケーションを形作る [24, 55]．クラスファイルとはプラットフォーム非依存のバイナリデータ (バイトコード) を含む最小の実行モジュールのことであり，Java VM[31] と呼ばれる仮想マシン上で動作する．Java 言語は Java VM の厳密な仕様により強力な逆コンパイラ [21, 27] が開発されており，ソフトウェアクラッカーは容易にクラスファイルを盗み出すことができる．すなわち，Java クラスファイルは使いやすく，守り難いといえる．

この問題に対処するため，この章では Java クラスファイルの盗用を発見する簡易な手法を提案する．具体的には，互いに非常によく似た (もしくは全く同一の) クラスファイルを発見するための手法，**バースマーク**を提案する．直感的に

バースマークとは，クラスが動作するために必要不可欠な特徴の集合のことを指す．そして，もし，クラス p と q が同じバースマークを持っていた場合，q は p のコピーである疑いが強いと言える (逆も然り).

高いスキルを持ったクラッカーは盗用の事実を隠すため，元のクラスファイルを何らかの手法で変更する可能性がある．そのため，理想的にはバースマークはプログラム変換に対する耐性を持つことが要求される．ゆえにバースマークは簡単に改ざんできるような特徴であってはならない．すなわち，バースマークを改ざんするとクラスが正常に動作しなくなったり，元のクラスとは全く異なる挙動になることが望ましい．一方，全く別の人物が同じ入出力仕様を持つクラスを別々に作成した場合，盗用ではないため，バースマークによって区別できることが望ましい．

これらの要件を考慮に入れ，我々は 4 種類のバースマークを提案する．それぞれ 変数初期値バースマーク，メソッド呼び出し系列バースマーク，継承関係バースマーク，使用クラスバースマークである．これらのバースマークは各クラスの基本的な特徴であり，ソースコードなしに抽出できる．提案手法に従ってバースマーク抽出ツール `jbirth` を開発した [56].

バースマークの有効性を確認するため，2 つの評価実験を行った．最初の実験では，バースマークのプログラム自動変換に対する耐性 (保存性) を評価した (第 5.2 節)．プログラム自動変換には Java 言語を対象とした難読化ツール，最適化ツール (CodeShield[13], Smokescreen[30], ZKM[65], jarg[40]) を利用した．バースマークの**類似度**の概念を導入することで，プログラムの自動変換に対して提案バースマークの耐性を示した．

次の実験では，バースマークの弁別性に対する評価を Java バイトコードを扱うことのできる 3 つのライブラリ BCEL[4]，javassist[10]，bloat[63] を対象に行った (第 5.3 節)．これらの著名なライブラリはオープンソースコミュニティの間で開発され，ソースコードは非常に多くの人に公開されているため，盗用に関係していないと考えられる．また，クラス設計も十分に練られていると考えられるため，重複した機能を持つクラスは存在しないと判断できる．そこで，各 jar ファイル内に重複したクラスがないと確認することで，提案バースマークの弁別性を

評価する．また，異なる作者が作成した仕様の似通ったライブラリに含まれるクラスは互いに異なることを確認するため，3 つの jar ファイルに含まれる全てのクラス間でバースマークを比較した．

議論において，現実的なバースマークの使用法について述べた．また，バースマークは，プログラムの静的解析により得られるものだけではなく [71, 57, 58, 59, 74]，プログラムの動的解析により得られるバースマークが提案されている [75, 68, 69, 60, 33, 34, 72]．これら 2 つのバースマークそれぞれの利点，欠点について述べ，プログラムの手動改変に対する提案バースマークの耐性について議論した．これらの実験と議論を通した結果から，提案バースマークは Java クラスファイルを発見するためのシンプルで，かつ強力な手法であることを示した．

2. 関連研究

盗まれたソフトウェアを証明する手段として，ソフトウェア電子透かし技術が良く知られている．ソフトウェア電子透かしはソフトウェアの著作情報やユーザの ID 番号のような情報をソフトウェアの一部に密かに埋め込んでおき，必要なときにその情報を取り出し，盗用の立証に役立てるものである．情報を埋め込む方法は 2 つに大別でき，プログラムの静的な部分に埋め込む方法 [18, 32, 67] やプログラムが動作するときの状態に埋め込む方法 [14, 15, 61] がある．しかし，電子透かしは，ソフトウェアの作成者が予め透かし情報を埋め込んでおく必要があり，過去に配布した (透かし情報の埋め込まれていない) ソフトウェアに対しては効力を持たない．加えて，ソフトウェアが複数のモジュールから構成される場合，ソフトウェアに含まれる全てのモジュールに対して透かし情報を埋め込む必要がある．これは，モジュールの数が多くなると現実的ではなく，また，全てのモジュールが透かしを埋め込むのに十分なサイズを持っているとは限らない．また，埋め込まれる情報はプログラムの実行という観点から見ると，無駄な情報であるため，最適化や難読化を施すことで削除されたり，人手によって改ざんされる可能性がある．

ソフトウェアのソースコードから類似度を測定し，ソフトウェアの盗用を発見

する方法も提案されている．これはプログラミング教育での盗用発見に用いられることが多い [1, 9, 45, 64]．プログラム教育での盗用とは，プログラムの提出を課された課題において，他人のプログラムをコピーし，プログラムの理解を必要としないような小さな変更，例えば変数名の付け替えを行って提出することである [42]．この盗用を発見するためにソフトウェアメトリクス [1, 45, 64] や Kolmogorov 複雑性 [9] を用いてプログラムの類似性を測る方法が提案されている．しかし，盗用されたソフトウェアの場合，ソースコードが手に入るとは考えにくい．また，これらの手法はソフトウェアの難読化や最適化などのツールによる攻撃を考慮していない．

コードクローンを用いてソフトウェアの盗用を発見する方法がある [6, 25]．コードクローンとはプログラムのソースコード中に存在する形や構造のよく似た部分のことを指す．もし，全く別のソフトウェアから大規模なクローンが発見された場合，どちらかのソフトウェアがコピーにより作られた疑いが非常に強い．神谷らはコードクローン発見ツール CCFinder を用いて FreeBSD の `sys/net/zlib.c` と Linux の `drivers/net/zlib.c` がほぼ同一であることを発見した [25]．コードクローンはコピーの疑いのあるプログラムを見つけるために有用であるが，やはり最適化や難読化などのプログラム変換による攻撃の影響を受け易い．例えば，ソフトウェアが盗用されたとき，ソースコードが手に入るとは考えにくいため，コードクローンを用いて盗用を見つけ出すには，バイナリのアセンブリ表現を用いて比較することになる．しかし，アセンブリ表現であると，コンパイルオプションの違いや，ソースコードの少しの変更により敏感に変わってしまう．そのため，コードクローンを用いて盗用を発見することは難しい．

最後に著作者解析 (authorship analysis) について述べる [29, 54]．[29] ではプログラミングスタイルメトリクスやプログラミングレイアウトメトリクスを用いてソースコードの著作者を特定する方法が提案されている．また，Spafford と Weeber は実行ファイル中からシステムコールやアルゴリズム，データ構造の選択といった著作者の作ったコードの断片を解析することでコンピュータウィルス，トロイの木馬の作者を特定する手法を提案している [54]．このようなソフトウェアが元来備える特徴から作者を特定する方法は Grover により **ソフトウェアバー**

スマーク と名付けられたが [20]，第 3 節で新たに形式的に定義した．今までに提案されているバースマークはやや古く，今日のプログラムに対しても有効であるかは不明である．ただし，我々が提案するバースマークには従来のバースマークを参考にしているものもある．

3. 準備

3.1 バースマークの定義

定義 1 (バースマーク) p, q を与えられたプログラムとする．$f(p)$ を p からある方法 f により抽出された特徴の集合とする．このとき，以下の条件を満たすならば，$f(p)$ を p の**バースマーク**であるという．

条件 1 $f(p)$ はプログラム p のみから得られる

条件 2 q が p のコピーであれば，$f(p) = f(q)$

条件 1 は，バースマークはプログラムの付加的な情報ではなく，p の実行に必要な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ことを表す．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．対偶から，もしバースマーク $f(p)$ と $f(q)$ が異なっていれば，q は p のコピーではないことを示す．

3.2 バースマークの満たすべき性質

バースマークは以下の性質を満たすことが望まれる．

性質 1 (保存性) p から任意の等価変換により得られた p' に対して，$f(p) = f(p')$ を満たす

性質 2 (弁別性) 同じ処理を行うプログラム p と q が全く独立に実装された場合，$f(p) \neq f(q)$ を満たす．

保存性はプログラム変換によりバースマークが変化しないことを表す．攻撃者は盗用の事実を隠すために，元のプログラムを等価な別のプログラムに変換することによりバースマークを改ざんする可能性がある．このような変換を行うための手法には，難読化 [13, 14, 16] や最適化があり，バースマークはこれらの攻撃によっても変化しないことが望ましい．

一方，弁別性は全く独立に実装されたプログラム p, q を正しく区別できることを表す．一般的にプログラムがある程度の規模であれば，プログラムの詳細な部分まで一致することは非常に稀である．しかしながら，たとえ p と q が全く独立に実装されていても，プログラムの規模が非常に小さい場合，この性質を満たさない場合がある．一般的にすべてのプログラムに対して，保存性・弁別性を完全に満たすバースマークを設計することは難しい．従って，実用上はユーザの判断により性質の強度を適宜決定する必要がある．

4. バースマーク

4.1 提案バースマーク

ここでは，4 種類のバースマークを提案する．それぞれ，**変数初期値バースマーク**，**メソッド呼び出し系列バースマーク**，**継承関係バースマーク** ，**使用クラスバースマーク**である．

図 2.1 に示す Java ソースコードを例にとり，それぞれのバースマークを説明する．ただし，図 2.1 のソースコードは説明の簡単化のために示すものである．我々の問題設定では，バイナリしか与えられることはなく，Java ソースコードは必要としない．

変数初期値バースマーク

Java 言語で書かれたクラスはフィールドと呼ばれる変数を持つ．これらの変数は宣言と同時に定数が代入されるとき，これをその変数の初期値とする．この初期値はオブジェクト生成における本質的な情報であり，初期値を変更するとプロ

```
 1: package jp.ac.aist_nara.se.tama.ant.taskdefs;
 2:
 3: import org.apache.tools.ant.Project;
 4:
 5: public class Echo extends org.apache.tools.ant.Task{
 6:   private String message = "";
 7:   private int level = Project.MSG_DEBUG; // 4
 8:   public void setMessage(String message){
 9:     this.message = message;
10:   }
11:   public String getMessage(){
12:     return message;
13:   }
14:   public void setLevel(String ll){
15:     ll = ll.toLowerCase();
16:     if(ll.equals("error"))        level = Project.MSG_ERR;     //0
17:     else if(ll.equals("warn"))    level = Project.MSG_WARN;    //1
18:     else if(ll.equals("info"))    level = Project.MSG_INFO;    //2
19:     else if(ll.equals("verbose")) level = Project.MSG_VERBOSE; //3
20:     else                          level = Project.MSG_DEBUG;   //4
21:   }
22:   public int getLevel(){
23:     return level;
24:   }
25:   public void execute() throws org.apache.tools.ant.BuildException{
26:     log(message, getLevel());
27:   }
28: }
```

図 2.1 Java プログラム例 (Apache Ant における echo タスク)

グラムの出力も変わる可能性がある．そのため，初期値はクラスの特徴であると言える．

定義 2 (変数初期値バースマーク) p を与えられたクラスファイルとし，クラス p で宣言されているフィールド変数を $v_1, v_2, ..., v_n$ とする．また，$t_i (1 \leq i \leq n)$ を v_i の型とし，$a_i (1 \leq i \leq n)$ を v_i の宣言時に代入されている定数値とする．もし a_i が現れなければ “null” とする．このとき，$((t_1, a_1), (t_2, a_2), ..., (t_n, a_n))$ を p の変数初期値バースマークと呼び，*CVFV(p)* で表す．

図 2.2 に図 2.1 に示すプログラムの変数初期値バースマークを示す．

メソッド呼び出し系列バースマーク

Java 言語では，J2SDK SE [55] や Jakarta プロジェクト [3] のような**既知のクラス**が数多くあり，それらクラスのメソッドを呼び出すことにより，目的のプログラムを作成する．そのため，メソッド呼び出しの順序は，以下の 2 つの理由からクラスを特徴付けるものであると言える．

最初の理由は，呼び出し順序を機械的に変更することは，メソッドの順序に依存関係があることも多くプログラムの誤作動の原因となることである．次の理由は，既知のクラスのメソッドを自ら用意し，系列を置き換えることは一般に困難であるものである．既知のクラスの多くは非常に複雑な処理を担当する．これは，既知のクラスのメソッドを置き換えるためには新たに既知のクラスを作り上げるのと同等の労力が必要となるため，実用的ではない．

定義 3 (メソッド呼び出し系列バースマーク) p を与えられたクラスファイルとし，C を与えられた既知のクラスの集合とする．$m_1, m_2, ..., m_n$ を p 中に現れる

```
CVFV(図2.1) = (
   (java.lang.String, ""),
   (int, 4)
)
```

図 2.2　図 2.1 に示すプログラムの変数初期値バースマーク

C に属するクラスのメソッド呼び出しを出現順に並べたものとする (実行順序ではない)．このとき，$(m_1, m_2, ..., m_n)$ を p のメソッド呼び出し系列バースマークと呼び，*SMC(p)* と表記する．

図 2.3 に図 2.1 に示すプログラムのメソッド呼び出し系列バースマークを示す．ただし，既知のクラスを J2SDK SE 1.4 に属するクラス，もしくは Jakarta プロジェクトに属するクラスとする．

継承関係バースマーク

オブジェクト指向言語である Java 言語において，全てのクラスは**クラス階層**と呼ばれる継承関係の構造を持っている．ただし，`java.lang.Object` は全てのクラスのルートとなるクラスであり，例外的に継承関係を持つことはない．そのため，与えられたクラス p からスーパークラスへと継承構造を辿ることによりクラスの系列が得られる．このクラスの順序はクラス p を特徴付けるものである．しかしながら，この順列には既知のクラスとユーザが作成したクラスが混在している．ユーザが作成したクラスを変更することは容易であるため，ユーザが作成したクラスを系列から削除したうえで，この系列をバースマークとして扱う．

定義 4 **(継承関係バースマーク)** p を与えられたクラス，C を与えられた既知のクラスの集合とする．$c_1, c_2, ..., c_n$ を次の条件を満たすような系列とする．

1. $c_0 = p$
2. $c_i (1 \leq i \leq n)$ は c_{i-1} のスーパークラス
3. c_n は継承ツリーの根のクラス (`java.lang.Object`)

もし，c_i が C に属していない場合，c_i を “null” に置き換える．そして得られた系列 $(c_1, c_2, ..., c_n)$ を継承関係バースマークと呼び，*IS(p)* と表記する．

図 2.4 に図 2.1 に示すプログラムの継承関係バースマークを示す．ただし，既知のクラスを J2SDK SE 1.4 に属するクラス，もしくは Jakarta プロジェクトに属するクラスとする．

```
SMC(図2.1) = (
   org.apache.tools.ant.Task(),
   String String#toLowerCase(),
   boolean java.lang.String#equals(Object),
   boolean java.lang.String#equals(Object),
   boolean java.lang.String#equals(Object),
   boolean java.lang.String#equals(Object),
   boolean java.lang.String#equals(Object),
   void org.apache.tools.ant.Task#log(String, int)
)
```

図 2.3 図 2.1 に示すプログラムのメソッド呼び出し系列バースマーク

```
IS(図2.1) = (
   org.apache.tools.ant.Task,
   org.apache.tools.ant.ProjectComponent,
   java.lang.Object,
)
```

図 2.4 2.1 に示すプログラムの継承関係バースマーク

使用クラスバースマーク

オブジェクト指向言語において，どのようなクラスであっても新しい機能の実現のために既存のクラスを使用する．**使用クラス**とは，あるクラス p のスーパークラス，メソッドの引数や返り値，またメソッドの呼び出しで現れるクラスのことを指す．p で使用しているクラスを変更することは，クラス間の処理の依存関係もあり，容易ではない．加えて，もしそれらの使用されるクラスが既知のクラスならば，改変はより難しくなる．従って，使用クラスは p を特徴付ける情報であるため，バースマークとして扱う．

定義 5 (使用クラスバースマーク) p を与えられたプログラム，C を与えられた既知のクラスの集合とする．U を p が使用しているクラスかつ C に属するクラスの集合とする．このとき，$u_1, u_2, ..., u_n (u_i \in U)$ を使用クラスバースマークと呼び，*UC(p)* と表す．

図2.5 に図2.1 に示すプログラムの使用クラスバースマークを示す．ただし，既知のクラスを J2SDK SE 1.4 に属するクラス，もしくは Jakarta プロジェクトに属するクラスとする．

```
UC(図2.1) = (
   java.lang.String,
   org.apache.tools.ant.Task,
   org.apache.tools.ant.Project,
   org.apache.tools.ant.BuildException,
)
```

図 2.5 2.1 に示すプログラムの使用クラスバースマーク

4.2 バースマークの類似度

プログラム p, q に対し，それぞれバースマーク $f(p) = (p_1, p_2, ..., p_n)$，$f(q) = (q_1, q_2, ..., q_n)$ が得られたとする．このとき，全ての i に対して $p_i = q_i$ である場合，$f(p)$ と $f(q)$ は**同一のバースマーク**であるといい，$f(p) = f(q)$ と書く．この場合，1つでも $p_i \neq q_i$ の組があれば，ほかの全てのペアが等しくても，$f(p) \neq f(q)$ となってしまう．よって，2つのバースマークが**非常に似ている**にもかかわらず，p と q は異なるプログラムであると結論付けてしまう．このように，全く同じかそうでないかでバースマークを比較すると，手法そのものが敏感になりすぎるため実用的ではない．この問題に対処するため，バースマークの**類似度**を定義する．

定義 6 (変数初期値バースマークの類似度) p と q を与えられたプログラム，p と q の 変数初期値バースマークを $CVFV(p) = ((tp_1, ap_1), (tp_2, ap_2), ..., (tp_n, ap_n))$，$CVFV(q) = ((tq_1, aq_1), (tq_2, aq_2), ..., (tq_m, aq_m))$ とする．また，k を $tp_i = tq_j$ かつ $ap_i = aq_j$ となるようなペアの数とする $(1 \leq i \leq n, 1 \leq j \leq m)$．このとき，$CVFV(p)$ と $CVFV(q)$ 間の類似度を $\frac{2k}{m+n}$ とする．

定義 7 (メソッド呼び出し系列バースマークの類似度) p と q を与えられたプログラム，p と q の メソッド呼び出し系列バースマークをそれぞれ $SMC(p) = (p_1, p_2, ..., p_n)$，$SMC(q) = (q_1, q_2, ..., q_m)$ とする．k を $p_i = q_i (1 \leq i \leq n, m)$ となるペアの数とする．このとき，$SMC(p)$ と $SMC(q)$ 間の類似度を $\frac{2k}{m+n}$ とする．

定義 8 (継承関係バースマークの類似度) p と q を与えられたプログラム，p と q の 継承関係バースマークを $IS(p) = (p_1, p_2, ..., p_n)$，$IS(q) = (q_1, q_2, ..., q_m)$ とする．k を $p_{n-i} = q_{m-i}$ となるような (p_{n-i}, q_{m-i}) のペアの数とする[1] $(1 \leq i \leq n, m)$．このとき，$UC(p)$ と $UC(q)$ 間の類似度を $\frac{2k}{n+m}$ とする．

定義 9 (使用クラスバースマークの類似度) p と q を与えられたプログラム，p と q の 使用クラスバースマークをそれぞれ $UC(p) = (p_1, p_2, ..., p_n)$，$UC(q) = (q_1, q_2, ..., q_m)$ とする．$UC(p)$ と $UC(q)$ の両方のバースマークに含まれている要素の数，すなわち，$p_i = q_j (1 \leq i \leq n, 1 \leq j \leq m)$ となる要素の数を k とする．このとき，$UC(p)$ と $UC(q)$ 間の類似度を $\frac{2k}{m+n}$ とする．

---

[1] $n - i, m - i$ としているのは，継承関係をルートから順に調べていくためである．

表 2.1 バースマーク抽出の実行時間

ライブラリ	クラス数	jar ファイルサイズ	実行時間
ant-1.6.1.jar	524	958,858 bytes	9.174s
bcel-5.1.jar	339	515,920 bytes	5.398s
bloat-1.0.jar	332	652,880 bytes	7.068s
javassist-3.0.jar	232	405,545 bytes	4.325s

5. 評価実験

5.1 jbirth

jbirthは提案バースマークを Java クラスファイルから抽出し，比較する我々が開発したツールである [56]．jbirthは Java (J2SDK SE 1.4[55]) で書かれ，BCEL 5.1[4] と共に動作する．図 2.6 に jbirthのスクリーンショットを示す．主な機能は以下の通りである．

- 4 種類のバースマークを Java クラスファイルから (ソースコードを参照せず) 直接抽出する．
- 抽出したバースマークを比較する．
- 将来的な拡張のためのプラグイン機構が導入されている．

パフォーマンス測定のため，標準的な Java のライブラリから 4 種類のバースマークを抽出する時間を計測した．測定は Pentium 4 3.00GHz, 496 MB RAM の Windows PC で行った．測定結果を表 2.1 に示す．この結果から，バースマークを抽出するために必要な時間は 1 クラス当たり約 0.018 秒と実用面から見ても十分に短いことがわかる．jbirthを用いることで，どのような Java のクラスからでもバースマークを簡単に抽出することができる．

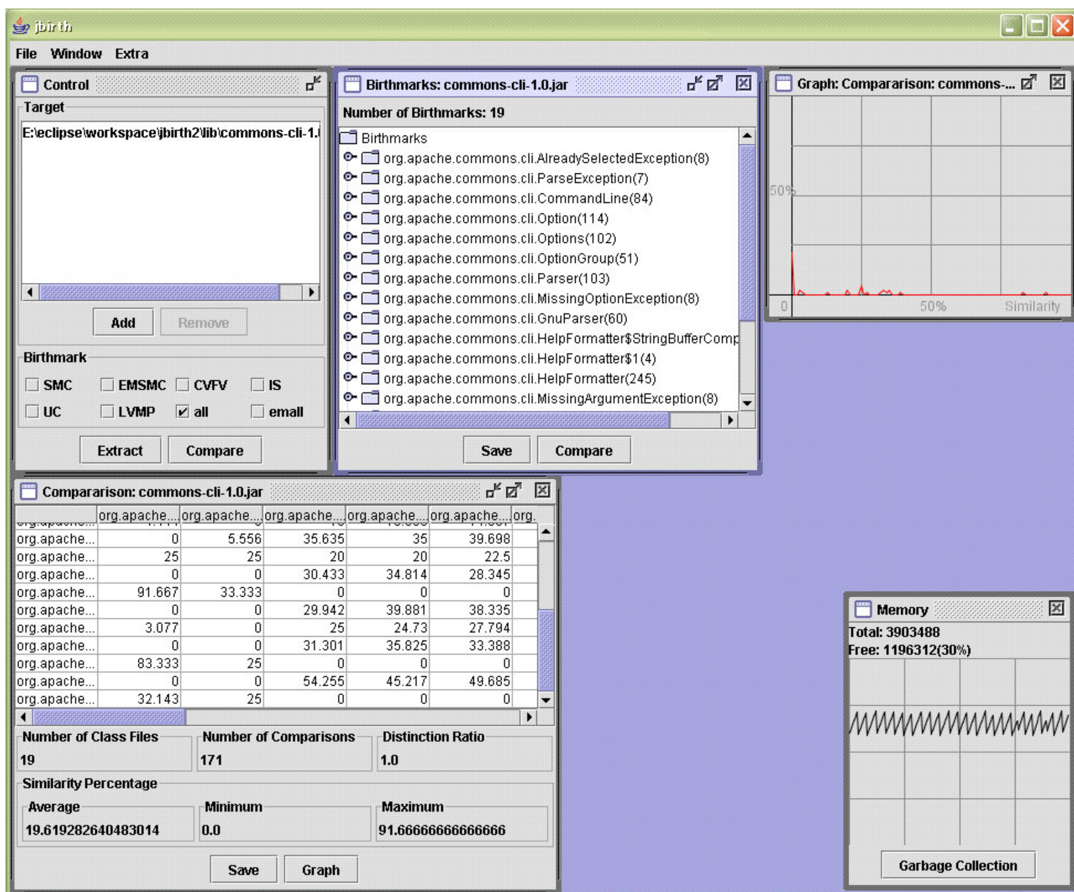

図 2.6 jbirth のスクリーンショット

5.2 保存性評価

最初に jbirth を用いて，提案バースマークの保存性の評価を行う (第 3 節 性質 1 参照)．知識のある攻撃者はプログラム p のバースマークを除去するために難読化や最適化を行うツールを用いて p と等価な仕様を持つ q を得るかもしれない．そこで，プログラム変換ツールにより，提案バースマークがどのくらい改ざんされるのかを確かめる．

この実験において，我々は ZKM[65]，Smokescreen[30]，CodeShiled[13]，jarg[40] の 4 つの Java プログラム変換ツールを採用した．いずれも Java クラスファイルを変換するためのツールであり，最初の 3 つは難読化を施すためのツールである．難読化とは，プログラムを外部仕様を保ったまま解析困難なものに変換する技術である．そして，jarg は最適化ツールであり，クラスファイルから実行に不必要な情報を削除する．

それぞれのツールのより具体的な機能について述べる．全てのツールは名前難読化やクラスファイル中に含まれるデバッグ情報の削除を行う．名前難読化とはクラス名やフィールド名などの名前 (シンボル名) を意味のある名称から意味のない名称に変換し，プログラムを理解し難いものにする手法である．また，ZKM, Smokescreen そして CodeShield は実行時の挙動を変更せずにコントロールフローを複雑に変換する機能を持つ．jarg と Smokescreen は実行されない部分や使われない変数やメソッドを削除する．ZKM は文字列暗号化という独特の機能を持つ．これは文字列定数を予め暗号化しておき，実行時に復号する処理を埋め込むことでプログラムを読みにくくする手法である．

実験の手順は，まず Apache Ant 1.6.1[2] に対して 4 つのツールを用いて最大限の難読化，最適化を施し，変換されたプログラムを得る．次に jbirth を用いて ant.jar に含まれる変換前と変換後の全てのペアの類似度を測定した．既知のクラスの集合は J2SDK SE 1.4 に含まれるクラスとした．

結果を表 2.2 に示す．ant.jar に含まれる 524 のクラスファイルのうち，インターフェースを除いた 487 クラスを提案バースマークそれぞれによって比較する．図 2.7 に類似度によるクラスの分布を表す図を示す．横軸はそれぞれのバースマーク，縦軸は頻度，奥行きは類似度を表す．この図より，多くのクラスにおいて，難読

表 2.2 実験 1 保存性評価の結果

		ZKM	Smokescreen	CodeShield	jarg
変数初期値バースマーク	**平均**	0.918	0.878	1.000	0.929
	最小	0.000	0.000	1.000	0.000
	最大	1.000	1.000	1.000	1.000
メソッド呼び出し系列バースマーク	**平均**	0.874	0.772	1.000	0.994
	最小	0.000	0.000	1.000	0.207
	最大	1.000	1.000	1.000	1.000
継承関係バースマーク	**平均**	1.000	1.000	1.000	1.000
	最小	1.000	1.000	1.000	1.000
	最大	1.000	1.000	1.000	1.000
使用クラスバースマーク	**平均**	0.984	0.993	0.714	0.980
	最小	0.000	0.000	0.000	0.667
	最大	1.000	1.000	0.975	1.000

化ツールを適用した後も提案バースマークは保存されていることがわかる．ただし，CodeShield で変換されたクラスに対する 使用クラスバースマークのみ，類似度 1.0 の数が他と比べ少なくなっている．これは CodeShield は if 文に代表される条件分岐を try, catch 節に変換し，ダミーの例外クラスを追加するためである．例えば，`if(EXP1){ s1; } else{ s2; }` という条件分岐は `try{ throw e; } catch(exp_1 e1){ s1; } catch(exp_2 e2){ s2; }` に変換される．このとき追加される例外クラスは `NullPointerException` や `IllegalArgumentException`, `IndexOutOfBoundsException` などの既知のクラスである．使用クラスバースマークバースマークはクラスが使用する既知のクラスの集合であるため，類似度が低くなると考えられる．そのため，使用された難読化アルゴリズムによっては既知のクラスの集合である C を調整しても良いだろう．

また，CodeShield 以外のツールにおいて，変数初期値バースマークの類似度が 0 になった組み合わせがいくつか存在した．原因を調査したところ，定数値が代入されているフィールドのいくつかが削除され，フィールドを使用していた箇

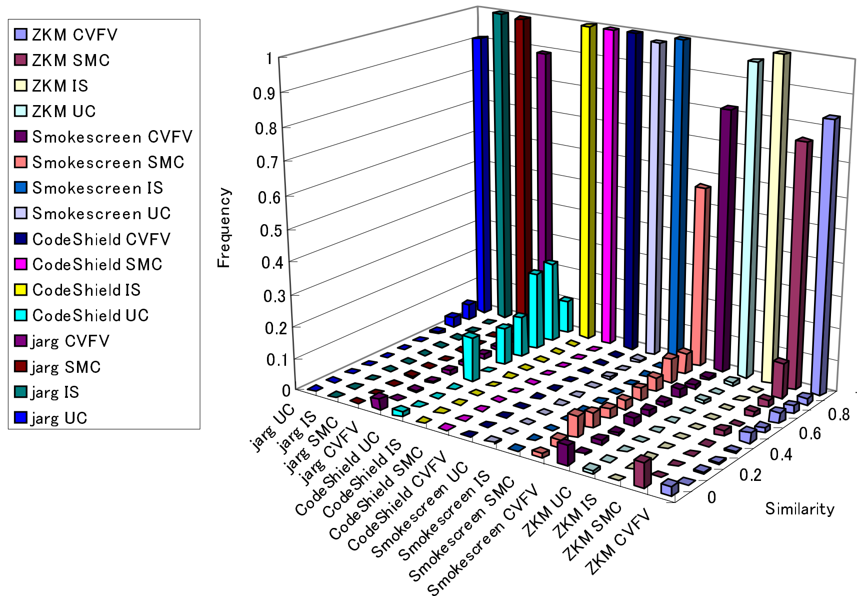

図 2.7 各バースマークの保存性評価 (各バースマーク毎)

所では，定数値を直接使用していることがわかった．また，どこからも参照されていないフィールドの場合は，そのフィールド自体削除されたため，変数初期値バースマークの類似度が一部低くなってしまったと考えられる．

このことから，単一のバースマークのみを持って比較するとそのバースマークの弱点を突き，攻撃が容易であると言える．そのため，バースマークでの比較は単一のバースマークで行うのではなく，複数のバースマークを同時に使用し，比較することが望ましい．提案した 4 つのバースマークを同時に使用したときの類似度の分布を図 2.8 に示す．このときの類似度は，それぞれのバースマークの類似度の平均とした．横軸が類似度を示し，縦軸は対応する類似度となったクラスのペアの数をクラス数で正規化したものである．図からわかるように，ほとんどの組み合わせで類似度が 0.8 以上になり，その中でも多くの組み合わせで，類似度が 1 になっている．このように，複数のバースマークを同時に用いることで一つのバースマークの弱点を回避することが可能である．

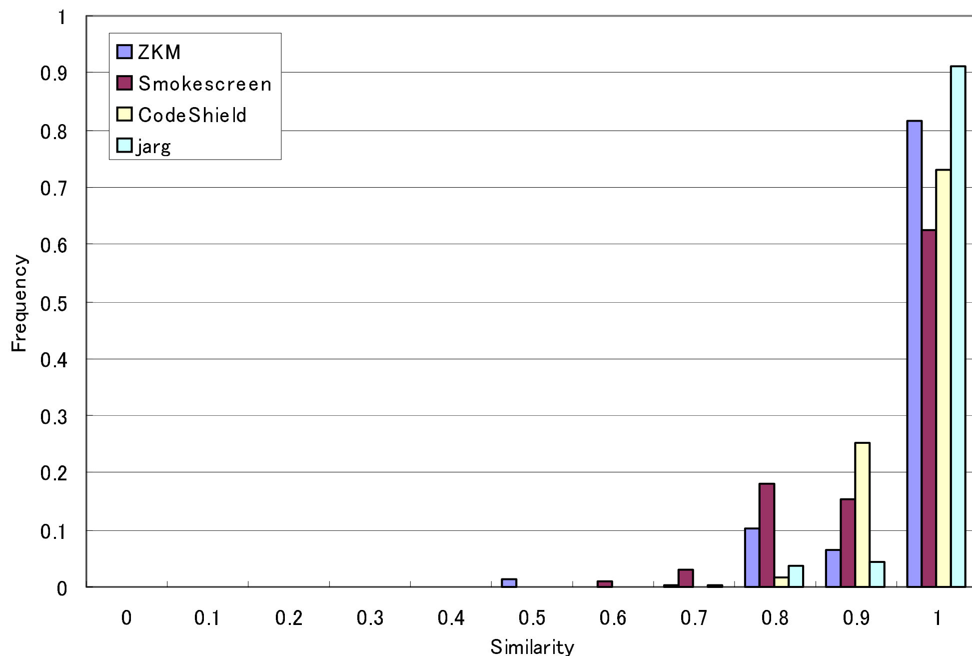

図 2.8 各バースマークの保存性評価 (4 つのバースマークを同時に使用)

5.3 弁別性評価

次に，提案バースマークの弁別性を評価する (第 3 節 性質 2 参照)．一つのアプリケーション中に，同じ機能を持つクラスが含まれていることは少ない．なぜなら，全く同じクラスが一つのパッケージに含まれているということは，クラスの設計に不備があると考えられるためである．それゆえ，多くの人に使われるようなアプリケーションは多くの人が様々な改善を施していくため，全く同じ，もしくは似たクラスも少なくなっていくものと考えられる．従って，一つのパッケージに含まれるいくつかのクラスをバースマークによって区別することで，弁別性を評価する．

評価を行ううえで，我々は BCEL 5.1[4], javassist 3.0[10], bloat 1.0[39] の 3 つのライブラリを選定した．3 つのライブラリは共に Java バイトコードを操作するためのものであり．BCEL は `jbirth` で使用しているライブラリである．また，javassist はアプリケーションサーバ JBoss[23] のアスペクト指向プログラミング

[26] のためのエンジンとして用いられており，bloat は sandmark[14] などに用いられている．それぞれのライブラリの jar ファイルに対して jbirth を実行させて，jar ファイルに含まれるバースマークを抽出し，jar ファイル内の全ての組み合わせでバースマークを比較した．また，3 つのライブラリは同じような用途で用いられているが，別々に開発されたため，同じバースマークを持つクラスは含まれていないと考えられる．そのため，3 つのライブラリの jar ファイルに含まれる全てのクラスのバースマークを相互に比較した．既知のクラスの集合は J2SDK SE 1.4 に含まれるクラスとした．

結果を表 2.3 に示す．平均，最小，最大はそれぞれのバースマークを用いたときの類似度を示している．そして，弁別率は，バースマークの類似度が 0.8 以上の場合，弁別できなかったとする条件で，どれくらいのクラスがバースマークで区別できたかを示している．また，類似度の分布を示したグラフを図 2.9 に示す．

表から 継承関係バースマークは他のバースマークに比べ弁別性が少し低くなっており，類似度の平均も少し高くなっていることがわかる．また，グラフからも継承関係バースマークは他のバースマークと比べ，類似度の高い結果が多く見られる．これは単純なクラスやユーティリティ的な役割のクラスの場合，java.lang.Object を直接継承することが多く，バースマークの要素数が少ないためだと考えられる．また，java.lang.Object を継承していないクラスであっても，同じスーパークラスを持つクラスが大量に存在する場合もある．例えば，BCEL に含まれるクラスのうち，org.apache.bcel.classfile.Attribute を直接継承するクラスは 13 個あり，org.apache.bcel.generic.ArithmeticInstruction を直接継承するクラスは 36 個存在する．また，BCEL において，変数初期値バースマークの弁別率が 0.666 と全ての評価の中で最低の値を示している．抽出された BCEL の変数初期値バースマークを調査すると長さ 0 のバースマーク (フィールド変数を持たないクラス) が多いことがわかった．

この結果はバースマークの単純性とのトレードオフであり，変数初期値バースマークや 継承関係バースマークをそのまま単独で用いるべきではないことを示す．変数初期値バースマークや 継承関係バースマークを用いるときには他のバースマークと併用することや，要素数の少ないバースマークは比較不可能であると

するなどの対策を講じる必要がある．

そこで，提案した 4 種類のバースマークを同時に用いて比較した場合の類似度の分布を図 2.10 に示す．横軸はバースマークの類似度を表し，縦軸は対応する類似度になったペアの数を比較回数で正規化したものである．図から多くのペアが 0.4 以下に集中している．ただし，BCEL において 0.9 以上の類似度となったペアがいくつか存在することがわかる．該当するペアの類似度を調査した結果，以下のようなクラスであることがわかった．

(1) 1 つか 2 つのメソッド呼び出ししか含んでいない小さなインナークラス (例えば，`System.out.exit(0)` のみ)

(2) 互いに区別できないようなほぼ同一のクラス (コピー&ペーストで作成されたと推測される)

上記の (1) は特徴付けするための情報を十分に持っていない小さなクラスであることを示す．このようなクラスの場合，もし，盗まれたとしても著作権を主張できるような知的財産を含めること自体困難であるため，保護する必要がない．もう一方では， ソースコードを調査した結果，コメントまで同じであったため，コピー&ペーストで作成されたクラスであった可能性が高く，似たクラスであった．そのペアの類似度が高くなったため，(2) は提案バースマークが十分実用的であることを示す．

6. 議論

6.1 現実的な提案手法の使用方法

バースマークの典型的な使用例はクラス p と q からそれぞれバースマークを抽出・比較して，q が p のコピーであるかを確認するものである．ただし，バースマークが一致したとしても，プログラムがコピーであるとは限らない (定義 1 より)．例え p と q が全く独立に作成されたとしても $f(p) = f(q)$ を満たす可能性は存在する．そのため，バースマークは盗用を証明する手段としては少し弱く，一つのバースマークが一致しただけで，盗用であると断定することはできない．

表 2.3 実験 2 弁別性評価の結果

		BCEL 5.1	javassist 3.0	bloat 1.0	全て
クラス数		339	222	320	881
比較回数		57,291	24,531	51,040	387,640
変数初期値バースマーク	平均	0.362	0.162	0.269	0.216
	最小	0.000	0.000	0.000	0.000
	最大	1.000	1.000	1.000	1.000
	弁別率	0.666	0.928	0.920	0.872
メソッド呼び出し系列バースマーク	平均	0.205	0.020	0.079	0.081
	最小	0.000	0.000	0.250	0.000
	最大	1.000	1.000	1.000	1.000
	弁別率	0.807	0.994	0.920	0.934
継承関係バースマーク	平均	0.681	0.607	0.717	0.632
	最小	0.143	0.250	0.167	0.143
	最大	1.000	1.000	1.000	1.000
	弁別率	0.702	0.825	0.681	0.780
使用クラスバースマーク	平均	0.235	0.165	0.233	0.151
	最小	0.000	0.000	0.000	0.000
	最大	1.000	1.000	1.000	1.000
	弁別率	0.860	0.995	0.945	0.962

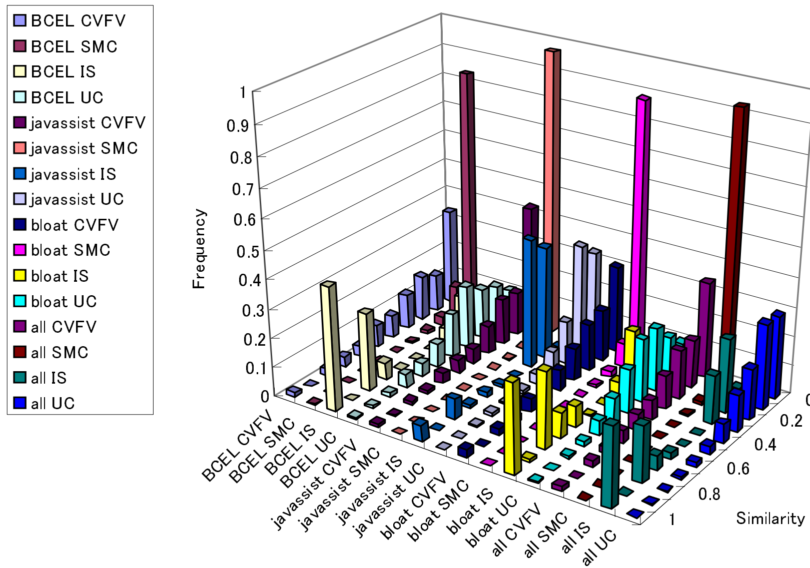

図 2.9 各バースマークの弁別性評価 (各バースマーク毎)

しかし，実用的な観点から言えば，バースマークは q が p のコピーであると証明する手掛かりの一つとして有用であると考えられる．実験 2 の結果を見れば，バースマークが偶然一致するクラスは，ほとんどの場合，単純で小さなクラスであった．そして，攻撃者が盗もうとするのは，単純で小さなクラスよりも，複雑で大きなクラスである場合が多いと考えられる．もし，p と q のサイズが十分に大きいのであれば，$f(p) = f(q)$ ならば q は p のコピーである可能性が高いと考えられる．従って，提案バースマークは大量のクラスファイルの中から盗用であると疑わしいクラスを見つけ出す手段として有効である．

そして，バースマークが実験 1 のように改ざんされにくい情報であれば，バースマークの一致だけを用いて盗用を発見するのではなく，バースマークの類似度を用いることで，より確実に盗用を発見することができるだろう．図2.8と図2.10の違いは非常に興味深いものである．難読化ツールにより改変したクラスのバースマークはオリジナルのバースマークと非常に良く似ており，互いに関係のないクラス間では非常に異なるバースマークが得られた．すなわち，類似度の適切な

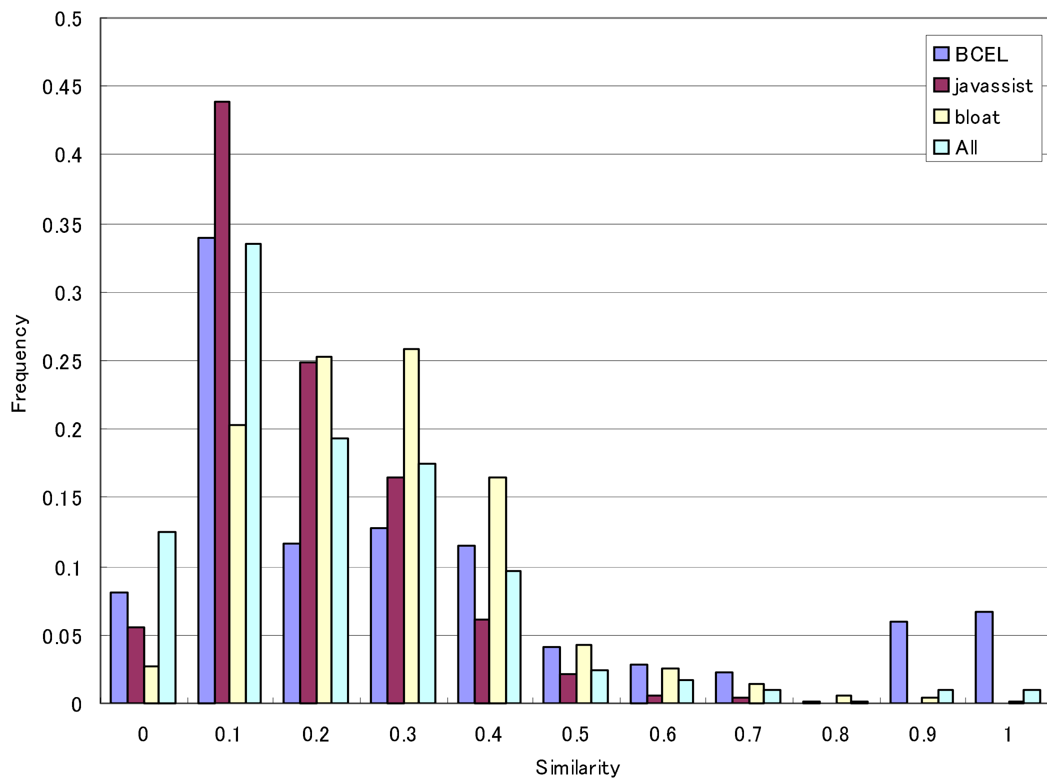

図 2.10 異なるクラス間でのバースマークの類似度 (4 つのバースマークを同時に使用)

閾値を与えることができれば提案手法はクラスのコピーを見つける手掛かりとし有効であると言える．

6.2 静的バースマークと動的バースマークの比較

以上で述べたバースマークはプログラムの静的解析により得られる特徴に基づいている．対して，プログラムの動的解析により得られる情報に基づいたバースマークもいくつか提案されている [33, 34, 68, 69, 75]．これらの動的バースマークは攻撃耐性，抽出容易性，評価単位の 3 つの点で静的バースマークと大きく異なっている．ここでは 3 つの点について，静的バースマーク，動的バースマークの違いを考察する．

まず最初に，攻撃耐性について述べる．静的バースマークはいくつかの攻撃に弱いことが指摘されている [74, 75]．静的バースマークはプログラムの静的解析により得られる情報を利用するため，バースマークとなる特徴が特定できれば，プログラムの意味解析を行うことで，比較的容易にバースマークを改ざんすることができる．対して，動的バースマークはプログラムの動的な情報を利用する．挙動を変えずにプログラムを改変することは難しいため，静的バースマークに比べて攻撃耐性があると言える．

次に抽出容易性について述べる．静的バースマークはプログラムの静的解析により抽出できると先に述べた．この作業は比較的高速に行うことができる．これに対して，動的バースマークを抽出するには，プログラムを実際に動作させ，入力を与えるという作業が必要である．この作業をある程度自動化することは可能であるが，実行に要する時間が必ず必要であるため，高速化することは難しい．また，GUI インターフェースを持つアプリケーションの場合，自動化は非常に難しい．このことから，一般的に静的バースマークは動的バースマークと比較して，抽出が容易である．また，静的バースマークは抽出が容易であるため，一度に多数のプログラムからバースマークを抽出し，抽出したそれぞれのバースマークを比較することが容易に実現できる．動的バースマークは抽出に手間がかかるため，このようなことは困難である．

最後に，バースマークの評価単位について述べる．提案する静的バースマーク

は，クラス単位で抽出する．対して，動的バースマークはほとんどがソフトウェア全体を単位として抽出する．そのため，静的バースマークの比較の単位はクラス単位となり，動的バースマークはソフトウェア単位となる．このことから，静的バースマークは動的バースマークに比べ細かい粒度で比較することができる．

以上のように静的バースマークと動的バースマークにはそれぞれ利点と欠点がある．ただし，静的バースマークと動的バースマークは競合する技術ではなく，互いに補い合う関係にある．電子透かしと異なり，バースマークは一つのプログラムにいくつでも共存させることが可能である．また，どのようなバースマークであっても常に偶然の一致の可能性があるため，実用的にソフトウェアの盗用を発見・証明するためには，静的バースマーク，動的バースマークの両方を含んだ複数のバースマークを併用することが重要であると考えられる．

6.3 手動改変に対する耐性

攻撃者がバースマークを手作業により改変しようとすることがある．例えば攻撃者が 変数初期値バースマークを改変しようと試みた場合，初期値を変更すれば良いため，代入されている定数を変更し，その後，本来の定数と変更後の変数の差を足す文を追加すれば 変数初期値バースマークを改変できる．すなわち，`int i = 5;` のような文を `int i = 3; i += 2;` と変更することで 変数初期値バースマークを改変することが可能である [74]．また，メソッド呼び出し系列バースマークも各メソッド呼び出しの間に何も行わないようなメソッド，例えば現在時間を取得するメソッドを呼び出すことで改変可能である．しかしながら，これらの脆弱性に対しては以下の理由からそれほど深刻な問題にならないと考える．

- Java バイトコードを手作業で変更するためには，非常に高度なスキルが必要となる．たとえ，そのスキルを持った攻撃者がいたとしても，元のコードの意味を正しく解釈し，バースマークを改変するには，多大な時間を要する．加えて，一つのクラスのバースマークを改変しただけでは，余り効果も得られないため，多くのクラスのバースマークを改変する必要があり，より大きな時間を必要とする．

- 実行効率を落とさずバースマークを消去することは一般に難しい．例えば，上に上げた 変数初期値バースマークを消去する方法や メソッド呼び出し系列バースマークを消去する方法の場合，確実に処理のオーバーヘッドが発生する．

- たとえ攻撃者が一つのバースマークを消去することに成功したとしても，別のバースマークは残っており，バースマークの類似度はそれほど低下しないと考えられる．そのため，手作業による改変は全てのコードに対して行う必要があり，時間的コストの面で実用的ではない．

このため，提案バースマークは簡潔で実用レベルで頑健な Java クラスファイルの特徴であると言える．

7. まとめ

この章では Java クラスファイルの盗用を発見するための方法であるバースマークを提案した．まず，プログラムバースマークについての定義を行い，4 種類のバースマーク，変数初期値バースマーク, メソッド呼び出し系列バースマーク, 継承関係バースマーク, 使用クラスバースマークを示した．

次に提案バースマークを 2 つの実験により評価した．我々は実験をバースマークの保存性，弁別性の 2 つの観点で行い，以下の結果を得た．

- 提案バースマークはプログラムの自動変換に対して強い耐性を持つ．すなわち，保存性を持つ．

- 小さなクラスを除き，多くの実用的なクラスにおいて，提案バースマークは弁別性を持つ．

また，バースマークについて，定性的な議論を行い，現実的な使用例，そして，プログラムの手動改変に対する耐性について述べた．

今後の課題として，提案バースマークのセキュリティについて，議論を深めること，そして，現実の攻撃者のバースマークに対する攻撃をモデル化し，実験を

行うことなどが挙げられる．また，他の新しいバースマークについても，引き続き調査を行っていく．

第3章　動的名前解決を用いたソフトウェア難読化

1. はじめに

インターネット等の情報流通基盤の発展により，ソフトウェアは入手しやすくなっている反面，その不正利用も年々深刻化している．**ソフトウェアプロテクション**とは，こうした不正利用からソフトウェア (含データ) を保護するための技術の総称であり，その重要性が高まってきている．

しかし，今日，ソフトウェアに適用されたプロテクトのほとんどが，プログラムの不正な解析行為 (**クラック**とよばれる) によって無効化されている．例えば，ある DVD 再生ソフトウェアがクラックされ，元来ライセンスを受けた開発ベンダしか知りえない，DVD コンテンツ復号鍵と復号アルゴリズムが漏洩したことは記憶に新しい [43]．その結果，DVD の暗号解除ツール，および，復号済みの DVD コンテンツがインターネット上に流出している．最近では，Apple 社の iTunes もクラックの対象となり，音楽コンテンツのコピープロテクトを無効化するパッチが公開されている [41]．

ソフトウェアをこうしたクラックから保護するため，**ソフトウェア難読化**が研究されている．難読化とは，与えられたプログラムを，その振舞いを変えないように，非常に読みにくいプログラムへと等価変換する技術であり，これまでにも様々な手法が提案されてきている (例：データ難読化 [17]，制御フロー難読化 [66]，演算難読化 [70, 73])．本論文では特に，オブジェクト指向ソフトウェアを対象に，**名前難読化** [62] に注目して議論を行う．

ソフトウェアをクラックする際の典型的な手法の一つとして，攻撃者がプログラム中の名前 (フィールド，メソッド，クラス，型等の識別子) を探し，それらを

頼りにプログラムの理解を行う方法がある．名前難読化は，これらの名前を意味のないわかりにくい名前へと変換し，元来の意味を隠蔽することで，攻撃者のクラックに必要なプログラム理解コストを増大させることを狙う．名前難読化は，実装が比較的容易なことから，現在ほとんどの難読化ツールに実装されている (例: DashO[46]，CodeShield[13]，ZKM[65])．

従来の名前難読化は，プログラム中の名前を別の名前に静的に置換するものである．したがって，自前で作成した (ユーザ定義の) メソッドやフィールド変数，クラス名などを任意の文字列に置換することは可能である．しかしながら，プログラムが利用するシステム API やライブラリ関数 (Java で例を挙げれば，`System.out.println()`) 等，システム定義の名前を静的に置換することは現実的に不可能である．あらゆる環境で汎用的に用いられる名前を変更することは，そのプログラムの移植性を著しく下げるためである．一般にこれらの名前は，プログラムの実行コード中にもそのまま現れ，攻撃者の格好の手がかりとなる [28]．

そこで本論文では，動的な名前解決を行うことでプログラム中の任意の名前を隠蔽する，新たな名前難読化手法を提案する．具体的には，まずプログラム中で隠蔽したい名前を事前に暗号化しておく．次に，`DynamicCaller` というクラスを用意して，それらの名前が関連する処理を全て `DynamicCaller` 経由で行うようにする．`DynamicCaller` は，暗号化された名前と処理内容 (インスタンス生成，メソッド実行，フィールド参照，または，フィールド代入) を文字列として受け取り，名前を復号した後，リフレクションを用いて動的に当該処理を呼び出す．これにより，本来の名前はソースコード，および，実行コード中には現れず，実行時にその名前が参照される直前にメモリ上にのみ現れることになる．処理終了後は，復号した名前を破棄することで，本来の名前が現れる時間を短くすることが出来る．提案手法により，従来不可能であったシステム API やライブラリメソッドの呼び出しも含め，あらゆる名前を隠蔽することが可能となる．結果として，プログラムの静的なクラックを著しく困難にすることが可能となる．

また，提案手法を Java プログラム用に実装し，性能およびセキュリティの観点から評価を行った．性能評価では，ある実用規模のプログラム (Jakarta Commons Digester 1.7) に対して難読化を行い，難読化前後の実行速度を比較した．その結

果，約 4 倍程度の性能劣化で難読化が可能となることが判った．セキュリティ評価では，静的解析と動的解析の 2 種類のクラック方法を想定して議論を行う．さらに，攻撃者が提案手法を知っている場合の攻撃に対する耐性についても議論を行う．

2. 準備

2.1 ソフトウェア難読化

ソフトウェア難読化とは，あるプログラムを非常に理解しづらい等価なプログラムに変換することで，プログラムを不正なクラックから保護するものである．難読化の概念をより厳密に定義するために，まず，プログラム理解について考える．人間 (攻撃者) がプログラムを理解する際には，様々な認知活動が行われるため，プログラム理解の一般的な定義を与えることは難しい．また，プログラムの何を理解するかで，認知活動は異なる．例えば，あるモジュールの機能を理解する場合と，データ構造を理解する場合，特定の関数の場所を理解する場合とでは，理解のプロセスが異なる．このことに注目し，本論文では，プログラムの理解とは「与えられたプログラムから理解の対象となる情報を取り出すこと」と定義する．

定義 10 (プログラム理解) p を与えられたプログラム，X を p に含まれる任意の情報 (の集合) とする．この時，ユーザが p から X を何らかの方法で外部に抽出 (リバース・エンジニアリング) できた時,「ユーザは p を X に関して理解した」と定義する．この時，ユーザが理解にかかるコストを $cost(p, X)$ と書くことにする．ここでコストとは，解析にかかる時間，労力，必要な知識，設備などを含む．

定義 11 (難読化) p を与えられたプログラム，X を与えられた p の情報 (の集合) とする．また，p の入出力写像を $\mathcal{IO}_p : I \rightarrow O$ と書く．ここで，I は全ての入力の集合，O は全ての出力の集合である．この時,「p の X に関する**難読化**」とは，あるプログラム変換 T を p に適用して，以下の条件を満たすプログラム $p'(= T(p))$ を得ることである．

条件 3 $\mathcal{IO}_p = \mathcal{IO}_{p'}$

条件 4 $cost(p, X) < cost(p', X)$

条件 3 は難読化前と難読化後のプログラムが，同一の入出力写像を持つ，すなわち，プログラムの外部的な振る舞いが変わらないことを保証するものである．条件 4 は，難読化後のプログラム p' から情報 X を取り出すことが困難になることを意味する．

難読化は必ずしも p のソースコードに適用されるとは限らない．一般に保護の対象となるプログラムのソースコードは公開されないため，むしろ，実行コード (例：ネイティブコードやバイトコード) やアセンブリコードに直接適用することが多い．

2.2 名前難読化

名前難読化は，プログラム中に現れる名前 (識別子) を別のものに付け替える難読化である．プログラム言語における名前は，計算機にとっては単なる識別子であるため，名前の付け替えがプログラムの挙動に影響を与えることはない．しかし，名前は人間にとってプログラムを理解するための重要な手がかりとなる [8]．したがって，元プログラムの名前を非常にわかりづらい名前に変換することで，難読化を実現する．

定義 12 (名前難読化) p を与えられたプログラム，U_p を p に現れる全ての名前の集合，$N_p (\subset U_p)$ を難読化すべき任意の名前の集合とする．p の**名前難読化**とは，p における各名前 $n \in N_p$ を別の名前 $n' (= T(n)$ とする) に変換し，別のプログラム p' を得る難読化である．ここで，T は一対一写像 $T : N_p \rightarrow N_{p'}$ ($N_{p'} \subset U_{p'}$, $N_p \cap N_{p'} = \phi$) である．

オブジェクト指向言語の場合，名前はプログラム p 中に，クラス名，フィールド名，メソッド名，変数名のいずれかの形で現れる．さらに，同じ名前でも**定義部** (宣言部) と**使用部** (呼び出し部) の双方に現れる．名前難読化では，これら p が含む全ての名前から，難読化すべき名前の集合 N_p をいかに選択するか，また，名前の変換方法 T をいかに実装するかが鍵となる．

2.3 従来手法

従来の名前難読化手法は，プログラム中の定義部に現れる名前を静的に別の名前に置き換えるものである．具体的には，以下の手順で行われる．
[従来の名前難読化手法]

入力: プログラム p，名前集合 N_p.

出力: 難読化されたプログラム p'.

手順: 各 $n \in N_p$ について，以下の操作を行う．結果として得られたプログラムを p' とする．

Step 1: 各 $n \in N_p$ について，p における n の定義部を別の名前 n' に置き換える．

Step 2: p における n の使用部を，Step 1 で置き換えた n' に置き換える．

入力における N_p として，ユーザ (開発者) が p で定義した任意の**ユーザ定義**の名前を与えることが出来る．図 3.1 にある Java プログラムを従来の名前難読化手法を用いて難読化した例を図 3.2 に示す[1]．

この例では，クラス名 `ImageViewer` を `a` に，フィールド名 `icon` を `aa` に，また，コンストラクタの引数の変数名 (ローカル変数名) を `aaa`，`myIcon` を `aaaa` にしている．また，`main` メソッドの引数を `aaa` に，ローカル変数 `i` を `aaaa` に，元 `ImageViewer` 型 (`a` に変更されている) のローカル変数を `aaaaa` に変更した[2]．

上で述べた手法は，比較的実装が簡単であり，難読化後のプログラムの性能劣化が少ないため，広く実用化されている．例えば，Java 言語用には，Dash-O[46], CodeShield[13]，ZKM[65] の他にも数多くの名前難読化ツールが存在する．

しかしながら，この手法は**システム定義**の名前に適用できないという大きな制限がある．システム定義の名前は汎用ライブラリや API クラスなどで定義される

[1] 議論の簡単にするため，ソースコードを用いて難読化を説明しているが，難読化のプロセスは必ずしもソースコードレベルで行われるとは限らない．

[2] Java 言語の場合，クラス名，メソッド名，フィールド名は異なる名前空間を使用しているため，同じ名前を使用しても問題なくコンパイル，実行を行うことも可能であるが，ここでは簡単化のため，異なる名前を使用している．

```
 1: import javax.swing.JFrame;
 2: import javax.swing.JLabel;
 3: import javax.swing.Icon;
 4: import javax.swing.ImageIcon;
 5: public class ImageViewer extends JFrame{
 6:   private Icon icon;
 7:   public ImageViewer(String file){
 8:     setTitle(file);
 9:     Icon myIcon = new ImageIcon(file);
10:     icon = myIcon;
11:     getContentPane().add(new JLabel(icon));
12:     pack();
13:   }
14:   public static void main(String[] args){
15:     for(int i = 0; i < args.length; i++){
16:       ImageViewer viewer = new ImageViewer(args[i]);
17:       viewer.setVisible(true);
18:     }
19:   }
20: }
```

図 3.1 サンプルプログラム (画像ビューワ)

名前であり，p においては使用部のみに現れる．システム定義の名前は，様々な環境で汎用的に利用される．したがって，p においてこれらを難読化してしまうと，p のポータビリティを著しく下げてしまうため，実用的ではない．図 3.2 の例では，JFrame や JLabel，setVisible，setTitle といった名前を別名に変更することはできない．これらは，javax.swing パッケージで定義される名前，また，JFrame で定義されているメソッド名で，変更すると別の環境で動作しなくなるからである．

このように，従来手法では，システム定義の名前を隠蔽できないため，攻撃者にプログラム理解の手がかりを残してしまうことになる．また，静的に置換された名前は，再置換も容易なため，難読化を解除されてしまうおそれがある．したがって，従来の名前難読手法は，難読化としてはあまり強力ではない．

```
 1: import javax.swing.JFrame;
 2: import javax.swing.JLabel;
 3: import javax.swing.Icon;
 4: import javax.swing.ImageIcon;
 5: public class a extends JFrame{
 6:   private Icon aa;
 7:   public a(String aaa){
 8:     setTitle(aaa);
 9:     Icon aaaa = new ImageIcon(aaa);
10:     aa = aaaa;
11:     getContentPane().add(new JLabel(aa));
12:     pack();
13:   }
14:   public static void main(String[] aaa){
15:     for(int aaaa = 0; aaaa < aaa.length; aaaa++){
16:       a aaaaa = new a(aaa[aaaa]);
17:       aaaaa.setVisible(true);
18:     }
19:   }
20: }
```

図 3.2 従来の名前難読化

3. 動的名前解決を用いた名前難読化

従来手法の問題点を解決すべく，システム定義の名前を難読化可能な新たな手法を提案する．

3.1 キーアイデア

第 2.3 節で述べたとおり，基本的にシステム定義の名前は変更すべきではないため，プログラム中から別名で呼び出す (使用する) ことはできない．よって，プログラム中の名前を静的に置換する従来の名前難読化法を適用することができなかった．これに対し，本研究では**動的名前解決**という機構を導入する．プログラム中で使用部に現れる名前をあらかじめ別の名前に変換 (暗号化) しておき，プロ

グラム実行時，その名前参照がある度に元の名前に戻す機構である．

多くのプログラミング言語では，関数 (メソッド) の呼び出しや変数参照に現れる名前は，コンパイル時に静的に決定している必要がある．しかしながら，オブジェクト指向言語には，**リフレクション**という機構が実装されており，実行時にクラスやメソッドの情報を取得することができる．クラスの情報を表示するアプリケーションや，また，プラグイン機構の実装など，利用するクラスの詳細が開発時にわからない場合に用いられる．リフレクションは情報を取得するだけではなく，インスタンスの型やフィールド，メソッド等を実行時に操作することもできる．これにより，動的に生成されたインスタンスから呼び出すメソッドの決定や，フィールドの操作を行うことができる．

本研究のキーアイデアは，リフレクションを用いることで，動的名前解決を行い，システム定義の名前を難読化することである．

3.2 動的名前解決

動的名前解決は，プログラム中の使用部に現れる名前を**文字列として**あらかじめ暗号化しておき，実行時，必要に応じて元の名前に戻すアプローチである．いま，難読化すべき名前の集合を N_p とする．ただし，N_p は使用部に現れる名前である．また，各 $n \in N_p$ に，あらかじめ暗号 E を適用し，得られた名前を $n'(= E(n))$ とする．

p において n は，(a) オブジェクト生成文におけるクラス名，(b) メソッド呼び出しにおけるメソッド名，(c) フィールド参照・代入におけるフィールド名 のいずれかに現れる．以下それぞれの場合について，どのように名前解決を行うのかを説明する．

オブジェクト生成文におけるクラス名

名前 n がオブジェクト生成文 (Java では new) のクラス名として現れる場合，以下のように名前解決を行う．ここで，$n' = E(n)$ とする．

手続き $resolveClass(n')$:

Step C-1: 文字列 n' を復号し，元のクラス名 n を得る．

Step C-2: リフレクションを用いて，文字列 n から名前が n であるクラス c を取得する．

Step C-3: c から新たなオブジェクト o を生成する．

メソッド呼び出しにおけるメソッド名

名前 n がクラス c のメソッド呼び出しに現れる場合，以下のように名前解決を行う．ここで，$n' = E(n)$，クラス c の名前が文字列 cn' で既に暗号化されているとする．

手続き $resolveMethod(cn', n')$ ：

Step M-1: 文字列 cn' に対して，$resolveClass(cn')$ のステップ C-1，C-2 を実行して，クラス c を取得する．

Step M-2: リフレクションを用いて，c が所有するメソッドの集合 M_c を取得する．

Step M-3: 文字列 n' を復号し，元のメソッド名 n を得る．

Step M-4: 名前が n であるメソッド m を M_c から取得する．

Step M-5: メソッド m を実行する．

フィールドの参照・代入におけるフィールド名

名前 n がクラス c のフィールド参照 (代入) のフィールド名として現れる場合，以下のように名前解決を行う．ここで，$n' = E(n)$，クラス c の名前が文字列 cn' で既に暗号化されているとする．

手続き $resolveField(cn', n')$ ：

Step F-1: 文字列 cn' に対して，$resolveClass(cn')$ のステップ C-1，C-2 を実行して，クラス c を取得する．

Step F-2: リフレクションを用いて，c が所有するフィールドの集合 F_c を取得する．

Step F-3: 文字列 n' を復号し，元のフィールド名 n を得る．

Step F-4: 名前が n であるフィールド f を F_c から取得する．

Step F-5: f に対して，参照または代入を行う．

3.3 動的名前解決を用いた名前難読化

提案する名前難読化は以下のとおりである．ただし，以下において N_p はシステム定義の名前を含んでもよい．
[提案する名前難読化手法]

入力: プログラム p，名前集合 N_p.

出力: 難読化されたプログラム p'.

手順: p に以下の手順を適用する．得られたプログラムを p' とする．

Step 1: 任意の文字列暗号 E を用いて，各名前 $n \in N_p$ を暗号化する．得られた集合を $N'_p = \{n' | n' = E(n)\}$ とする．

Step 2: 各 $n \in N_p$ について，p における n の使用部を以下のように置き換える．

n **がクラス名の場合:** $resolveClass(n')$ を実現するコードに置き換える．

n **がメソッド名の場合:** $resolveMethod(cn', n')$ を実現するコードに置き換える．

n **がフィールド名の場合:** $resolveField(cn', n')$ を実現するコードに置き換える．

```
import javax.swing.JFrame;
        :
public class ImageViewer extends JFrame{
  private Icon icon;
  public ImageViewer(String file){
    setTitle(file);
    Icon myIcon = new ImageIcon(file);
        :
```

↓ setTitle と ImageIcon を難読化

```
import javax.swing.JFrame;
        :
import java.lang.Class;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
public class ImageViewer extends JFrame{
  public javax.swing.Icon icon;
  public ImageViewer(String file){
    // resolveMethod("JnbhfWjfxfs", "tfuUjumf")
    Class c1 = Class.forName(decrypt("JnbhfWjfxfs"));
    Method m = c1.getMethod(decrypt("tfuUjumf"),
                            new Class[]{ file.getClass() });
    m.invoke(this, new Object[]{ file });

    // resolveClass("kbwby/txjoh/JnbhfJdpo")
    Class c2 = Class.forName(decrypt("kbwby/txjoh/JnbhfJdpo"));
    Constructor c = c2.getConstructor(new Class[]{ file.getClass() });
    Object o2 = c.newInstance(new Object[]{ file });
        :
```

図 3.3 動的名前解決を用いた難読化 (図 3.1 のコード)

図 3.3 に，提案手法による名前難読化例を Java コードで示す．このコードは，図 3.1 におけるメソッド呼び出し (`setTitle`) とオブジェクト生成 (`ImageIcon`) を動的名前解決によって難読化した例である．上記 2 つの名前は，汎用クラス `JFrame` で定義されるシステム定義の名前である．簡単のため，名前文字列の暗号化に鍵 1 のシーザー暗号を用いている．コード中の `decrypt()` は復号ルーチンを示す．

最初のブロックでは，動的名前解決 resolveMethod により `setTitle` のメソッド呼び出しを実現している．Java 言語では，`java.lang.Class` クラスがクラス情報を反映するクラスに相当し，`forName` メソッドに文字列を与えることでクラスを生成できる (ステップ C-2)．また，`java.lang.reflect` パッケージにメソッドやフィールドなどを反映するクラスが含まれる．これらを使うことでリフレクション機能を用いることができ，生成されたクラスから，そのクラスが持つメソッドやフィールドの情報を得ることができる (ステップ M-4, F-4)．この例では，“tfuUjumf” を復号した文字列 “setTitle” からメソッドを取得し，invoke により実行している (ステップ M-5)．次のブロックでは，`ImageIcon` オブジェクトの生成を，resolveClass() により行っている．`java.lang.Class` クラスを用いて，文字列 “javax.swing.ImageIcon” からクラスを生成し，`newInstance` メソッドによってオブジェクトを生成している (ステップ C-3)．難読化後のコードでは，オリジナルコードのシステム定義名が完全に隠蔽されていることがわかる．

4. 実装

提案する難読化手法を自動化するため，Java プログラムを対象とする難読化ツールの実装を行った．実装したツールは，任意の Java クラスファイル (バイトコード) を入力とし，クラス内の全ての名前参照を難読化したクラスファイルを出力する．ここでは，実装の詳細について述べる．

表 3.1 DynamicCaller のメソッド

メソッドインターフェース	対応する処理
Object newInstance(Object[] arg, String EclassName)	*resolveClass*
Object invoke(Object instance, Object[] arg, String EclassName, String EmethodName) Object invokeStatic(Object[] arg, String EclassName, String EmethodName)	*resolveMethod*
Object getField(Object instance, String EclassName, String EfieldName) Object getStatic(String EclassName, String EfieldName)	*resolveField* (reference)
void setField(Object instance, Object value, String EclassName, String EfieldName) void setStatic(Object value, String EclassName, String EfieldName)	*resolveField* (assignment)

4.1 動的名前解決支援クラス DynamicCaller の導入

動的名前解決を支援するため，本ツールでは DynamicCaller という新たなクラスを実装した．このクラスは，表 3.1 に示す 7 つのメソッドを持ち，難読化対象のプログラム p に組み込まれる．

表中の各メソッドは，第 3.2 節で提案した動的名前解決の手続きをラップする．暗号化されたシステム定義の名前文字列（EclassName, EmethodName, または EfieldName）を引数にとり，該当する動的名前解決を行う．また，オブジェクト生成またはメソッド呼び出しの際に渡される引数は，Object 型の配列 arg を通して渡される．Static の接尾語がついているメソッドは，static メンバを扱う場合のメソッドである．この場合，操作対象のオブジェクト (インスタンス) を省略可能である．

また，DynamicCaller は文字列暗号の復号ルーチンを内包する．現在の実装では DES[35], AES[38], DES-EDE [37], MD5[48] をサポートしている．

4.2 クラスファイルの書き換え

難読化対象プログラム p の Java クラスファイルが与えられると，本ツールは以下の手順で p の難読化を行う．

準備

まず，p に含まれる全クラスのメンバ (フィールド，メソッド) に対して，アクセス権を public に書き換え，外部クラスからアクセス可能にする．これは，動的名前解決を p の外部クラス `DynamicCaller` を用いて行うためである．

また，p における全てのフィールドとローカル変数 (仮引数以外) の型を `Object` 型に書き換える．`Object` は Java 言語における全てのクラスの親クラスにあたるため，p の動作を変えることなく，p に現れる型情報を隠蔽することができる．この書き換えは，動的名前解決の手法そのものには関係なく，より解析を困難にするための付加的な操作である．例えば，図 3.1 の 6, 9 行目の型 `Icon` は，`Object` に変換して差し支えない．

名前を暗号化する

p において難読化すべき名前を，与えられた暗号アルゴリズムと鍵を用いて暗号化する．これは，第 3.3 節で提案する難読化手順の Step 1 に相当する．このとき用いる暗号は，`DynamicCaller` における復号ルーチンで用いる暗号と合致させておく必要がある．本ツールは，p のクラスファイル内の Constant Pool (定数データを保存するために確保される領域) を検索し，クラス名，メソッド名，フィールド名を取得する．次に，各名前について暗号化を行った後，得られた名前を文字列 (String) として新たに Constant Pool に追加する．

`DynamicCaller` の呼び出し部を埋め込む

p における静的な名前参照を，`DynamicCaller` を用いて動的に名前解決するように，クラスファイルを書き換える．これは，第 3.3 節で提案する難読化手順の

Step 2 に相当する．

クラスファイルの書き換えは，図 3.4 に示す書き換えルールに従って行われる．各ルールは，Java の擬似バイトコードを用いて書かれている．図の左列がオリジナルのバイトコード，右列が変換後のバイトコードである．図中，p において難読化する元の名前を Foo，fooMethod，fooField，これらを暗号化した名前をそれぞれ Bar，barMethod，barField と仮定している．名前が使用される場所によって，(a), (b), (c) または (d) の変換規則を適用する．

図 3.4(a) は，Foo がオブジェクト生成時のクラス名として現れる場合である．元のバイトコードでは，Foo オブジェクトが生成された後，コンストラクタへの引数をスタックに積み，invokeSpecial によって Foo オブジェクトのコンストラクタが実行される．変換後のバイトコードでは，まずスタックに詰まれた引数を一度配列に格納して，Foo の暗号化文字列である"Bar"をスタックに積み，DynamicCaller の newInstance メソッド (表 3.1 参照) を呼びだす．元のバイトコードではオブジェクトの生成とコンストラクタの呼び出しは別のステップで行われていたが，変換後のバイトコードではオブジェクト生成とコンストラクタの呼び出しは同時に行われる．

スタック上の引数を配列に格納する理由は，これらの引数を DynamicCaller に配列として渡す必要があるからである．図 3.5 に，スタック上に詰まれた 2 つの引数 arg1, arg2 を配列に格納するための，バイトコード列およびオペランドスタックの状態遷移を示す[3]．

同様に，図 3.4 の (b), (c), (d) ではそれぞれ，メソッド呼び出しの場合，フィールド参照の場合，フィールド代入の場合のバイトコード書き換えルールを示している．図から static メンバに関する変換ルールは割愛しているが，インスタンスをスタックに積む処理がない (最初の *push instance to stack* が存在しない) だけでほぼ同様の手順で変換を行う．

[3] 引数がプリミティブ型の場合は，aastore を実行する直前にラッパクラスに置き換えるバイトコードを挿入する必要がある．また，引数が double 型や long 型ならばオペランドスタック上で 2 ワードの領域を使うため，dup2_x1 ではなく，dup2_x2 としなければならない．

	Original bytecode	**Obfuscated bytecode**

(a) Object instantiation	:
	`new Foo`
	`dup`
	push args to stack
	`invokespecial Foo#<init>`
	:
		:
		push args to stack
		store args to array
		`ldc "Bar"`
		`invokestatic`
		`DynamicCaller#newInstance`
		:

(b) Method invocation	:
	push instance to stack
	push args to stack
	`invokevirtual Foo#fooMethod`
	:
		:
		push instance to stack
		push args to stack
		store args to array
		`ldc "Bar"`
		`ldc "barMethod"`
		`invokestatic`
		`DynamicCaller#invoke`
		:

(c) Field reference	:
	push instance to stack
	`getfield Foo#fooField`
	:
		:
		push instance to stack
		`ldc "Bar"`
		`ldc "barField"`
		`invokestatic`
		`DynamicCaller#getField`
		:

(c) Field assignment	:
	push instance to stack
	push value to stack
	`putfield Foo#fooField`
	:
		:
		push instance to stack
		push value to stack
		`ldc "Bar"`
		`ldc "barField"`
		`invokestatic`
		`DynamicCaller#setField`
		:

図 3.4 バイトコード書き換えルール

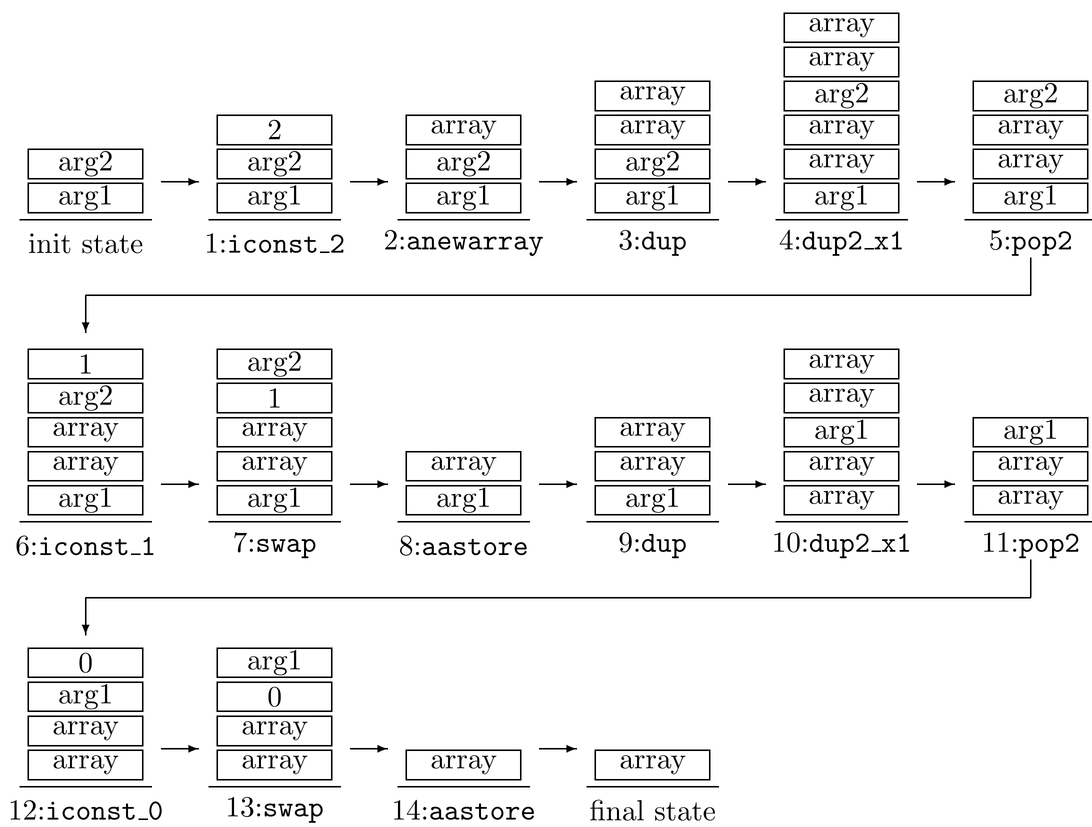

図 3.5 スタック上の引数を配列に格納する手順

4.3 難読化例

図 3.1 のプログラムに対し実装したツールを適用して難読化を行った．結果を図 3.6 に示す．この例では，文字列暗号に 56 ビット DES を用いた．復号の鍵は 0xefb08fa71fdcc29e である．なお，説明の簡単化のため，図 3.6 には Java のソースコードを示しているが，ツールではクラスファイルを直接書き換えることに注意されたい[4]．

図 3.6 のコードでは，オリジナルコード (図 3.1) で使用された全てのシステム定義の名前が難読化されている．例えば，図 3.6 の 4 行目からの一文は，図 3.1 の setTitle(file) に相当する．このとき，図 3.6 のコードのみを静的に解析して，setTitle(file) が実行されることを理解するのは，ほぼ不可能である．同様に，getContentPane.add(), ImageIcon, Icon といったシステム定義名も完全にコードから隠蔽されている．さらに動的名前解決は，ユーザ定義のクラス，メソッドの使用部 (定義部は不可) に対しても適用できるため，図 3.1 の main メソッドも同様に難読化することができる．

また，ソースコード上に現れるクラス名は JFrame，ImageViewer，Object，String，DynamicCaller，Integer，Boolean だけである．Object 型は全てのクラスのルートとなるクラスであり，提案難読化手法は全ての型を Object 型で表すため当然のことながら出現する．そして，ImageViewer とそのスーパークラスである JFrame，また，3, 24 行目の String はクラスやメソッドの宣言部に現れている．この部分はメソッドの定義部であり，提案難読化手法の対象外である．また，30 行目の Integer と 34 行目の Boolean はラッパクラスへの変換のために必要な呼び出しであり，この部分を DynamicCaller 経由で実行することは無意味である．なぜなら，DynamicCaller 経由で実行したとしても，DynamicCaller.newInstance(new Object[] { new Integer(10) }, "java.lang.Integer") のようになるだけで，全く余計な処理を行うことになるためである．加えて，25 行目のキャストで出現する Integer も隠すことが不可能である．これは比較演算子が対象とするのは int 型の数値であり，Object 型で

[4] o1, o2 への代入は説明の簡単化のために加えたもので，クラスファイル上では代入は行われていない

はない．そのため，`Object` 型の変数を，int 型に変換する必要がある．そのとき `Integer` クラスで実装されている `intValue` を呼び出す必要があるが，`Object` 型の変数に対して直接呼び出すことはできない．キャストを行うことで初めて `intValue` を呼び出すことができるため，ここの `Integer` を隠すことはできない．

5. 評価

5.1 性能評価

提案手法では，静的な名前参照を全て動的な呼び出しに変換するため，難読化後のプログラムの実行速度の低下が懸念される．そこで，実用規模のプログラムを難読化し，難読化によるオーバーヘッドを評価した．

難読化の対象として Jakarta Commons Digester 1.7[5] を採用した．これは XML で書かれたデータファイルを読み込み，そのデータをメンバとして持つ Java オブジェクトを生成するための汎用ツールである．比較的簡単に扱えることもあり，多くの Java アプリケーションにおいて設定ファイルの読み込み等に使われている．

実験では，難読化したプログラムに Digester チュートリアル [22] の XML ファイルを読み込ませ，オブジェクトが作成されるまでの時間を測定する．測定環境は，Windows XP Professional SP2 (Intel Pentium 4 3.00GHz，1GB RAM) である．文字列の暗号化には 56 ビット DES を採用した．

実験手順は Digester に含まれる 55 のクラスのうち，ある割合だけランダムに選択して難読化する．こうして得られたクラスファイル群に対し，実行時間を 10 回計測し，その平均を算出することを 10 回繰り返した．難読化するクラスの割合は 0%, 25%, 50%, 75%, 100% とした．難読化に要した時間を表 3.2 に示す（単位はミリ秒）．

表 3.2 難読化に要した時間

難読化の割合	25%	50%	75%	100%
所要時間 (ミリ秒)	2501	2758	3145	3539

```
 1:public class ImageViewer extends JFrame{
 2:  public Object icon;
 3:  public ImageViewer(String file){
 4:    DynamicCaller.invoke(this, new Object[]{ file },
 5:      "f29f6e89c20c525ca9ad991e2806eec6",
 6:      "e46a1b932f90efc76d5606a286a3c585");
 7:    Object myIcon = DynamicCaller.newInstance(new Object[]{ file },
 8:      "2309ad079c732c3eee007fd16630de66fbe9154c7e632f85");
 9:    DynamicCaller.setField(this, myIcon,
10:      "f29f6e89c20c525ca9ad991e2806eec6", "6c25f9e31afd88e4");
11:    Object o1 = DynamicCaller.invoke(this, new Object[0],
12:      "f29f6e89c20c525ca9ad991e2806eec6",
13:      "2ff8414ab86e8975a6439f43d8690008");
14:    Object o2 = DynamicCaller.newInstance(new Object[]{
15:      DynamicCaller.getField(this, "f29f6e89c20c525ca9ad991e2806eec6",
16:      "6c25f9e31afd88e4") },
17:      "2309ad079c732c3ea3d8e7d5ada803b78542f3f87b7db5c4");
18:    DynamicCaller.invoke(o1, new Object[]{ o2 },
19:      "5459bd4f08f7b753d035fd5b82780cd28aa49371f35be2ea",
20:      "878dfc57666c5eb9");
21:    DynamicCaller.invoke(this,new Object[0],
22:      "f29f6e89c20c525ca9ad991e2806eec6", "bd627b43d640f316");
23:  }
24:  public static void main(String[] args){
25:    for(int i = 0; i < ((Integer)DynamicCaller.invokeStatic(
26:        new Object[]{ args },
27:          "f127001edc194f1a034a6c9b64b22b57326fe908709c0456",
28:          "637ff5714bb80cc1cd5952a95803c6ea")).intValue(); i++){
29:      Object o1 = DynamicCaller.invokeStatic(new Object[]{ args, new
30:        Integer(i)}, "f127001edc194f1a034a6c9b64b22b57326fe908709c0456",
31:        "669658105ab0e482");
32:      Object viewer = DynamicCaller.newInstance(new Object[]{ o1 },
33:        "f29f6e89c20c525ca9ad991e2806eec6");
34:      DynamicCaller.invoke(viewer, new Object[]{ new Boolean(true) },
35:        "f29f6e89c20c525ca9ad991e2806eec6",
36:        "8f98baa6837f9236b117be0cb0a26503");
37:    }
38:  }
39:}
```

図 3.6 難読化後のサンプルプログラム

表 3.3 難読化したプログラムの DynamicCaller 呼び出し回数

	0%	**25%**	**50%**	**75%**	**100%**
DynamicCaller#newInstance	0	44.6	127.7	161.7	197
DynamicCaller#invoke	0	1814.8	3457.5	5136.1	7114
DynamicCaller#invokeStatic	0	56.5	90.6	159.3	237
DynamicCaller#getField	0	558.3	1349.2	1889.9	2329
DynamicCaller#getStatic	0	6.2	9.1	15.2	30
DynamicCaller#setField	0	135.8	377.8	537.2	671
DynamicCaller#setStatic	0	0.6	0.7	1.2	2
DynamicCaller 呼び出し総数	0	2616.8	5412.6	7900.6	10580

実験結果を表 3.3, 表 3.4 に示す．表 3.3 は，難読化後のプログラムにおいて，DynamicCaller の各メソッドがそれぞれ平均何回呼ばれたか，またそれらの合計回数を示している．また，表 3.4 には，難読化後のプログラムの実行時間の平均，中央値，最大，最小 (それぞれ単位はミリ秒) を示している．最下段には，難読化後のプログラムのサイズ (全てのクラスファイルの合計サイズ，単位はバイト) を示す．また，図 3.7 は難読化後のプログラムの実行時間を表す．

全てのクラスを難読化した場合 (100%) の実行時間の平均は，元プログラム (0%) の 4.11 倍に増加している．DynamicCaller のメソッドの呼び出し回数が多いほど，実行時間が増加し，かつ，ファイルサイズは 2.20 倍に増加している．

したがって，ハードリアルタイム性が要求されるプログラムや，実行環境に容量制限がある場合には，名前にセキュリティ上の優先順位をつけ，難読化を行うべき名前の集合 N_p を絞りこむことが望ましい．また，従来の静的な名前難読化を併用し，ユーザ定義の名前に関しては従来法で難読化するといったアプローチも可能である．

表 3.4 難読化したプログラムの実行時間とファイルサイズ

		0%	25%	50%	75%	100%
実行時間 (ミリ秒)	平均値	348.1	985.0	1143.7	1280.4	1429.7
	中央値	347.5	982.9	1121.2	1376.2	1414.5
	最大値	353.0	1128.5	1365.6	1417.6	1484.0
	最小値	346.0	822.0	878.1	965.1	1406.0
ファイルサイズ (byte)		165,831	247,533	281,221	307,815	365,314

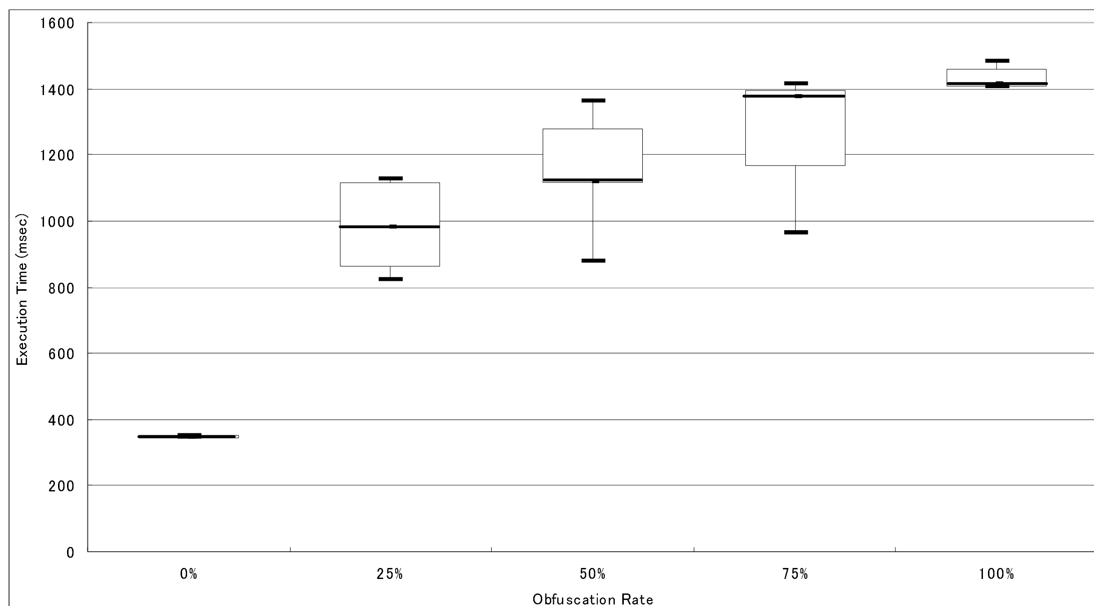

図 3.7 Digster の実行結果

5.2 セキュリティ評価

プログラム解析による攻撃を想定し，提案手法のセキュリティ評価を行う．前提として，攻撃者は難読化後のプログラム p' しか入手できないものとする．

まず**静的解析**に対する難読化の耐性について考察する．静的解析は，プログラムを実行せずにプログラム内の静的な情報のみを解析する攻撃である．提案手法で難読化されたプログラムでは，オブジェクト生成，メソッド呼び出し，フィールド参照・代入が全て動的呼び出しに変更されている．さらに，呼び出しに現れる名前は全て暗号文字列で置き換えられている．したがって，逆アセンブルや逆コンパイル等のリバースエンジニアリングを行っても，暗号化されたシンボル名しか抽出されない．これらの暗号文字列を復号をするのは暗号の解読と同じ労力が必要となる．さらに，たとえ暗号が解読されたとしても，全ての変数の型が`Object`型に置き換えられているため，どのクラスのメンバが呼び出されるのかをプログラムを実行せずに一意に決定することが難しい．このことから，この難読化手法は静的解析に非常に強いことが言える．

次に，デバッガ等を用いてプログラムを実行させ，実行中のプログラム情報を解析する**動的解析**による攻撃を考える．具体的には，実行時のある時点でプログラムを停止 (ブレーク) させ，スタックトレースを出力する攻撃が考えられる．この攻撃により，その時点でスタック上にある全てのメソッド名が出力される．これらのメソッドは`DynamicCaller`から動的に呼び出された後，スタックに積まれるため，オリジナルのメソッド名が露出してしまう．そのため，提案難読化手法は動的解析に対して脆弱である．対策としては，アンチデバッガ技術等，動的解析そのものを防ぐアプローチを取る必要がある．

また，`DynamicCaller`に含まれる復号ルーチンおよび復号鍵は，ソフトウェアが配布された後，変更することは難しい．そのため，ある暗号化された文字列と元の名前の対応付けが行われると，同じ暗号化文字列が現れる他の部分の元の名前も対応付けられてしまう．また，暗号化された名前の頻度を解析することで，元の名前を推測する攻撃方法が考えられる．これを解決するために，各名前ごとに異なる鍵を用いて復号するようにし，同じ暗号化文字列がプログラム中に出現しないようにする方法が考えられる．

最後に，攻撃者がDynamicCallerを解析することが想定される．DynamicCallerは暗号化文字列の復号ルーチン (および復号鍵) を含むため，攻撃者がこれらを理解すると難読化を解除する手がかりを与えてしまう．したがって，DynamicCallerの内容を何らかの方法で隠蔽することが望ましい．残念ながら，提案手法によって DynamicCaller そのものを難読化することはできない．動的名前解決が再帰的になるからである．したがって，white-box 暗号 [11, 12] 等の別の難読化手法を採用する必要がある．また，DynamicCaller は，与えられたプログラムに依存しない汎用クラスであるのでハードウェアとして実装するといった対策も考えられる．DynamicCaller の効果的な保護方法については，今後の検討課題としたい．

5.3 他の名前難読化手法との比較

図 3.8 に，従来の名前難読化法 (2.3 参照) と提案法の適用範囲を示す．従来法では，基本的に名前の定義部を書き換えるため，システム定義のクラス (の使用) を隠蔽する目的に使えなかった．提案法では，名前の使用部を暗号化し動的呼び出しを行うことで，この問題を解決している．なお，従来法と提案法は併用が可能である．その場合は，まず従来法によってユーザ定義の名前を難読化した後，提案法でシステム定義の名前 (使用部) を難読化することになる．

関連研究として，プログラム全体を暗号化しておき実行の直前に復号する**プログラム暗号化**が存在する [36]．提案手法との違いは秘密情報がメモリ上に現れている時間である．プログラム暗号化では，プログラムが実行されている間，復号されたプログラム全体がメモリ上に展開される．対して提案手法では，名前参照の直前に復号し，呼び出しが終了すると復号されたデータを破棄するため，メモリ上にオリジナルの情報が現れる期間が極めて短い．

また，刑部らはオブジェクト指向言語の多態性 (ポリモーフィズム) を用いて，異なるメソッドを同一名にする難読化手法を提案している [49]．この方法も名前難読化の一つに数えられるが，システム定義の名前がオーバーライドされている場合に適用できない．

			従来の名前難読化
		システム定義の名前	ユーザ定義の名前
	名前定義部	クラス定義	クラス定義
	名前定義部	メソッド定義	メソッド定義
	名前定義部	フィールド定義	フィールド定義
提案難読化手法	名前使用部	変数宣言	変数宣言
提案難読化手法	名前使用部	オブジェクト生成	オブジェクト生成
提案難読化手法	名前使用部	メソッド呼び出し	メソッド呼び出し
提案難読化手法	名前使用部	フィールド参照	フィールド参照
提案難読化手法	名前使用部	フィールド代入	フィールド代入

図 3.8　従来法と提案法の適用範囲

6. まとめ

本論文では，プログラムの静的解析が困難になるよう，動的名前解決を用いてプログラムを難読化する方法を提案した．提案手法では，あらゆるメソッド呼び出し，フィールド参照・代入を暗号化した名称でアクセスするため，プログラムを静的に解析することを劇的に困難にすることができる．提案難読化手法によるパフォーマンスの劣化を測定した結果，4.11 倍の低下に抑えることができた．最後に，`DynamicCaller` を効果的な保護する方法の考案を今後の課題とする．

第4章　終わりに

本論文では，ソフトウェアの開発と利用の両局面において，ソフトウェアを保護するための手法を検討した．第2章において，開発の局面におけるソフトウェアの不正利用を防止するため，バースマークの概念を提案し，定義を行った．バースマークを用いることで，ソフトウェアの盗用を効果的に発見することができる．また，応用として，自社ソフトウェアがオープンソースのソースコードを組み込んでいないことを提案バースマークにより確認することもできる．

ソフトウェアバースマークは，ソフトウェアに対して何も処理を行わず，特徴となる情報を抽出するだけなので，一つのソフトウェアに複数のバースマークを制限無く共存させせることが可能である．そのため，多くのバースマークを用いることで，盗用発見の精度を上げることができると考えられる．また，従来ソフトウェアの盗用を証明・発見する技術として研究されてきた電子透かしとも競合することはない．従って，電子透かしとも併用して用いることが現実的であろう．

次に，第3章では，ソフトウェアを利用する局面における不正利用を防ぐため，動的名前解決を用いたソフトウェア難読化手法を提案した．提案手法を用いることで，静的解析においては，隠蔽したいメソッド呼び出しを完全に隠すことができる．これにより，認証処理などの呼び出し箇所を悪意あるユーザの解析から隠蔽することが可能になり，エンドユーザサイドにおけるソフトウェアの不正利用を防ぐことができる．

もし，悪意ある開発者がオープンソースソフトウェアの盗用を隠すため，提案難読化手法を用いた場合，提案バースマークのうち，変数初期値バースマーク，メソッド呼び出し系列バースマーク，使用クラスバースマークのほとんどが削除される．しかし，これについては次の理由から現実的ではないと考える．盗用するソフトウェアの一部のみに提案難読化手法を用いた場合，難読化適用部分以外

の部分において，似たバースマークが検出されることが考えられるためである．加えて，提案難読化手法は，実行速度の劣化が著しいため，バースマークを改変するためだけの目的で，盗用するソフトウェア全体に提案難読化手法を適用するのも現実的ではないと考えられるためである．

謝辞

本研究を進めるに当たり，研究方法などに関する多くのアドバイスと共に丁寧なご指導を賜りました，奈良先端科学技術大学院大学情報科学研究科 ソフトウェア工学講座 松本 健一 教授に深謝致します．

本研究を進めるに当たり，的確で貴重なご助言を頂きました 奈良先端科学技術大学院大学情報科学研究科 情報基礎学講座 関 浩之 教授に深謝致します．

本研究を進めるに当たり，本研究に対し，有益な提案を頂きました 奈良先端科学技術大学院大学情報科学研究科 情報基礎学講座 楫 勇一 助教授に深謝致します．

本研究を進めるに当たり，多くの技術的なアドバイスに加え，細部にわたる熱心なご指導を頂きました，奈良先端科学技術大学院大学情報科学研究科 ソフトウェア工学講座 門田 暁人 助教授に深謝します．

本研究を進めるに当たり，多くのアドバイスと共に研究の整理や論文作成など，熱心で丁寧なご指導を頂きました，奈良先端科学技術大学院大学情報科学研究科 ソフトウェア工学講座 中村 匡秀 助手に，心から深く感謝します．

本研究を進めるに当たり，多くのアドバイスを頂きました 奈良先端科学技術大学院大学情報科学研究科 ソフトウェア工学講座 大平 雅雄 助手に，心から深く感謝します．

本研究を進めるに当たり，多くのアドバイスを頂きました 奈良先端科学技術大学院大学情報科学研究科 ソフトウェア設計学講座 飯田 元 教授に深く感謝します．

本研究を進めるに当たり，活発な議論を行い，研究の糧となった 奈良先端科学技術大学院大学情報科学研究科 ソフトウェア工学講座 井垣 宏 特任助手に感謝します．

本研究を進めるに当たり，多くの技術的なアドバイスを頂いた 奈良先端科学技術大学院大学情報科学研究科 ソフトウェア工学講座 神崎 雄一郎 氏に感謝します．

本研究を進めるに当たり，奈良先端科学技術大学院大学情報科学研究科 ソフトウェア工学講座 ソフトウェアセキュリティグループのメンバの岡本 圭司 君，山

内 寛己 君の 2 人には技術的なご助言やご協力を頂きました．ここに記して謝意を表します.

本研究を進めるに当たり，発表資料準備などにおいて，ご助言やご協力を頂いた 奈良先端科学技術大学院大学情報科学研究科 ソフトウェア工学講座の皆様に感謝します.

最後に，今までの人生を支えてくれた両親，祖父母，親戚に感謝します.

本研究の一部は EASE プロジェクト (Empirical Approach to Software Engineering) の援助を受けています.

参考文献

[1] Alex Aiken. MOSS: A system for detecting software plagiarism, June 2004. `http://www.cs.berkeley.edu/~aiken/moss.html`.

[2] Apache Software Foundation. Apache Ant. `http://ant.apache.org/`.

[3] Apache Software Foundation. Apache Jakarta project. `http://jakarta.apache.org/`.

[4] Apache Software Foundation. Jakarta BCEL, October 2001. `http://jakarta.apache.org/bcel/`.

[5] Apache Software Foundation. Jakarta commons digester, June 2005.

[6] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM: the International Conference on Software Maintenance*, pp. 368–377, November 1998.

[7] Business Software Alliance. Second annual BSA and IDC global software piracy study, May 2005.

[8] B. D. Chaudhary and H. V. Sahasrabuddhe. Meaningfulness as a factor of program complexity. In *Proc. of the ACM 1980 annual conference*, pp. 457–466, 1980.

[9] Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. SID plagiarism detection, December 2003. `http://genome.math.uwaterloo.ca/SID/`.

[10] Shigeru Chiba. Javassist: Java bytecode engineering made simple. *Java Developer's Journal*, Vol. 9, issue 1, January 2004.

[11] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an aes implementation. In *9th Annual International Workshop on Selected Areas in Cryptography, SAC 2002, Lecture Notes In Computer Science*, Vol. 2595, pp. 250–270, August 2002. Newfoundland, Canada.

[12] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. A white-box DES implementation for DRM applications. In *2nd ACM Workshop on Digital Rights Management (DRM2002), Lecture Notes in Computer Science*, Vol. 2696, pp. 1–15, November 2003. Washington, DC, USA.

[13] CodingArt Pty Ltd. Codeshield: Java byte code obfuscator, 1999. `http://www.codingart.com/codeshield.html`.

[14] Christian Collberg. Sandmark: A tool for the study of software protection algorithms, 2000. `http://www.cs.arizona.edu/sandmark/`.

[15] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Proc. Principles of Programming Languages 1999, POPL'99*, pp. 311–324, January 1999. San Antonio, TX.

[16] Christian Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, Vol. 28, No. 8, pp. 735–746, August 2002.

[17] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *Proc. 1998 International Conference on Computer Languages*, pp. 28–38, Washington D.C., USA, May 1998. IEEE Computer Society.

[18] Robert L. Davidson and Nathan Myhrvold. Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884, September 1996. Filed: June 30, 1994.

[19] Free Software Foundation, Inc. GNU general public license version 2, 1991. `http://www.gnu.org/copyleft/gpl.html`.

[20] Derrick Grover, editor. *The protection of computer software —its technology and applications Second edition*. The British Computer Society Monographs in Informatics Cambridge University Press, May 1992.

[21] Jochen Hoenicke. JODE: Java optimize and decompile environment, 1998. `http://jode.sourceforge.net/`.

[22] Philipp K. Janert. Learning and using Jakarta digester, October 2002. `http://www.onjava.com/pub/a/onjava/2002/10/23/digester.html`.

[23] JBoss Group LLC. JBoss. `http://www.jboss.com/`.

[24] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley Pub Co, June 2000.

[25] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. on Software Engineering*, Vol. 28, No. 7, pp. 654–670, July 2002.

[26] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect oriented programming. In *Proc. European Conference on Object-Oriented Programming (ECOOP) (Springer-Verlag LNCS)*, Vol. 1241, June 1997.

[27] Pavel Kouznetsov. jad - the fast java decompiler, Feburary 2004. `http://www.kpdus.com/jad.html`.

[28] Kracker's and BEAMZ. クラッカー・プログラム大全一禁断のシリアルナンバー解析テクニック. データハウス, December 2003.

[29] Ivan Krsul and Eugene H. Spafford. Authorship analysis: identifying the author of a program. *Computers and Security*, Vol. 16, No. 3, pp. 233–257, 1997.

[30] Lee Software. Smokescreen Java obfuscator, 2000. `http://www.leesw.com/`.

[31] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification Second Edition*. Addison-Wesley Pub Co, April 1999.

[32] Akito Monden. jmark: A lightweight tool for watermarking java class files, 2002. `http://se.aist-nara.ac.jp/jmark/`.

[33] Ginger Myles and Christian Collberg. Detecting software theft via whole program path birthmarks. In *Proc. Information Security 7th International Conference, ISC 2004*, Vol. 3225, pp. 404–415. Springer-Verlag GmbH, September 2004. Palo Alto, CA, USA.

[34] Ginger Myles and Christian Collberg. K-gram based software birthmarks. In *Proc. the 2005 ACM symposium on Applied computing*, pp. 314–318, March 2005. Santa Fe, New Mexico.

[35] NBS (National Bureau of Standards). Data encryption standard (DES). Technical Report FIPS-Pub.46, National Bureau of Standards, U.S. Department of Commerce, Washington D.C., January 1977.

[36] New-Era-Soft. JEncoder, 2005. `http://www.new-era-soft.com/`.

[37] NIST (National Institute of Standards and Technology). Data encryption standard, 1999. `http://csrc.nist.gov/cryptval/des/fr990115.htm`.

[38] NIST (National Institute of Standards and Technology). Advanced encryption standard (AES), March 2001.

[39] Nathaniel John Nystrom. Bytecode-level analysis and optimization of Java classes. Master's thesis, Faculty of Purdue University, August 1998.

[40] Hidetoshi Ohuchi. jarg - java archiver grinder, January 2003. http://jarg.sourceforge.net/index.en.

[41] Andrew Orlowski. DVD jon unlocks iTunes' locked music. The Register, November 2003. http://www.theregister.co.uk/2003/11/22/dvd_jon_unlocks_itunes_locked/.

[42] Alan Parker and Hamblen O. James. Computer algorithms for plagiarism detection. *IEEE Transactions on Education*, Vol. 32, No. 2, pp. 94–99, May 1989.

[43] Andy Patrizio. DVD piracy: It can be done. Wired News, November 1999. http://www.wired.com/news/technology/0,1282,32249,00.html.

[44] Andy Patrizio. Why the DVD hack was a cinch. Wired News, November 1999.
http://www.wired.com/news/technology/0,1282,032263,00.html.

[45] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, Vol. 8, No. 11, pp. 1016–1038, November 2002.

[46] PreEmptive Solutions. DashO - the premier Java obfuscator and efficiency enhancing tool.
http://www.agtech.co.jp/products/preemptive/dasho/index.html.

[47] Eric Raymond and Rob Landley. OSI position paper on the SCO-vs.-IBM complaint, May 2004. http://www.opensource.org/sco-vs-ibm.html.

[48] Ronald L. Rivest. The MD5 message-digest algorithm. Technical Report RFC 1321, MIT LCS and RSA Data Security, Inc., April 1992. http://www.rfc-editor.org/rfc/rfc1321.txt.

[49] Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Java obfuscation —approaches to construct tamper resistant object-oriented programs. *IPSJ Journal, Special Issue on Research on Computer Security Characterized in the Context of Social Responsibilities*, Vol. 46, No. 8, pp. 2107–2119, August 2005.

[50] Ryan Singel. Cherry OS re-released, still fishy. Wired News, March 2005. `http://www.wired.com/news/mac/0,2125,66847,00.html`.

[51] 萌え系フリーウェアの国際的盗用事件. slashdot.jp, September 2001. `http://slashdot.jp/article.pl?sid=01/09/09/0744206`.

[52] Epson pulls linux software following GPL violations. slashdot.org, September 2002. `http://slashdot.org/article.pl?sid=02/09/11/2225212`.

[53] Daniel Smith, Michael Militzer, Christoph Lampert, and Edouard Gomez. Xvid team requests sigma designs' to halt copyright infringement, August 2002. `http://www.xvid.org/press/press-20020822-en.pdf`.

[54] Eugene H. Spafford and Stephen A. Weeber. Software forensics: Can we track code to its authors? *Computers and Security*, Vol. 12, No. 6, pp. 585–595, October 1993.

[55] Sun Microsystems., Inc. Java 2 SDK standard edition. `http://java.sun.com/se/`.

[56] Haruaki Tamada. jbirth: A tool for extracting birthmarks from Java class files, 2003. `http://se.aist-nara.ac.jp/jbirth/`.

[57] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Detecting the theft of programs using birthmarks. Information Science Technical Report NAIST-IS-TR2003014 ISSN 0919-9527, Graduate School of Information Science, Nara Institute of Science and Technology, November 2003.

[58] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. 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, Feburary 2004. Innsbruck, Austria.

[59] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Java birthmarks —detecting the software theft—. *IEICE Transactions on Information and Systems*, Vol. E88-D, No. 9, pp. 2148–2158, September 2005.

[60] Haruaki Tamada, Keiji Okamoto, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Dynamic software birthmarks to detect the theft of windows applications. In *Proc. International Symposium on Future Software Technology 2004 (ISFST 2004)*, October 2004. Xi'an, China.

[61] Clark Thomborson, Jasvir Nagra, Ram Somaraju, and Charles He. Tamper-proofing software watermarks. In *Proc. 2nd workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, Vol. 32, pp. 27–36, January 2004. Dunedin, New Zealand.

[62] Paul M. Tyma. Method for renaming identifiers of a computer program. United States Patent 6,102,966, August 2000. Filed: March 20, 1998, Issued: August 15, 2000.

[63] David Michael Whitlock. The bloat book, October 1998. `http://www.cs.purdue.edu/s3/projects/bloat/`

[64] Michael J. Wise. YAP3: Improved detection of similarities in computer program and other texts. In *Proc. 27 SIGCSE technical symposium on Computer science education*, pp. 130–134, December 1996. Philadelphia, Pennsylvania, United States.

[65] Zelix Pty Ltd. Zelix Klass Master, 1997. `http://www.zelix.com/klassmaster/index.html`.

[66] 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. 電子情報通信学会論文誌 D-I, Vol. J80-D-I, No. 7, pp. 644–652, July 1997.

[67] 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, pp. 3001–3009, November 2000.

[68] 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提案. ソフトウェア工学の基礎 XI, 日本ソフトウェア科学会 (FOSE 2004), pp. 85–88. 近代科学社, November 2004.

[69] 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. 第 46 回プログラミング・シンポジウム報告集, pp. 41–50, January 2005.

[70] 佐藤弘紹, 門田暁人, 松本健一. データの符号化と演算子の変換によるプログラムの難読化手法. 電子情報通信学会技術報告 情報理論研究会 (情報通信サブソサイエティ合同研究会), 第 IT2002-49 巻, pp. 13–18, March 2002.

[71] 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の提案. 電子情報通信学会 技術研究報告 情報セキュリティ研究会, 第 ISEC 2003-29 巻, pp. 127–133, July 2003.

[72] 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, Vol. J88-D1, No. 10, pp. 1595–1599, October 2005.

[73] 福島和英, 清水晋作, 田中俊昭. 多変数の符号化による難読化方式の提案. コンピュータセキュリティシンポジウム 2004 (CSS 2004) 予稿集, pp. 247–252, October 2004.

[74] 福島和英, 田端利宏, 櫻井幸一. Java バイトコードの静的解析によるプログラム指紋の抽出方法. 情報処理学会研究報告, コンピュータセキュリティ研

究会 2003-126, 第 126 巻, pp. 81–86, December 2003.

[75] 福島和英, 田端利宏, 櫻井幸一. 動的解析を用いた java クラスファイルのプログラム指紋抽出法. 暗号と情報セキュリティシンポジウム 2004 (SCIS 2004), 第 1 巻, pp. 1327–1332, January 2004.

付録

A. バースマークの規模

A.1 ソースコード行数に対するバースマークの要素数の分布

バースマークの全ての要素の分布

図 A.1〜A.4 に示すグラフは第 2 章バースマークの実験（第 5 節）に用いた各ライブラリに含まれるクラスファイルの行数と，抽出されたバースマークの要素数の関係を示したものである．各グラフの横軸はコメント行，空行を除いたソースコードの行数を表し，縦軸は各バースマークの要素数を対数スケールで表したものである．

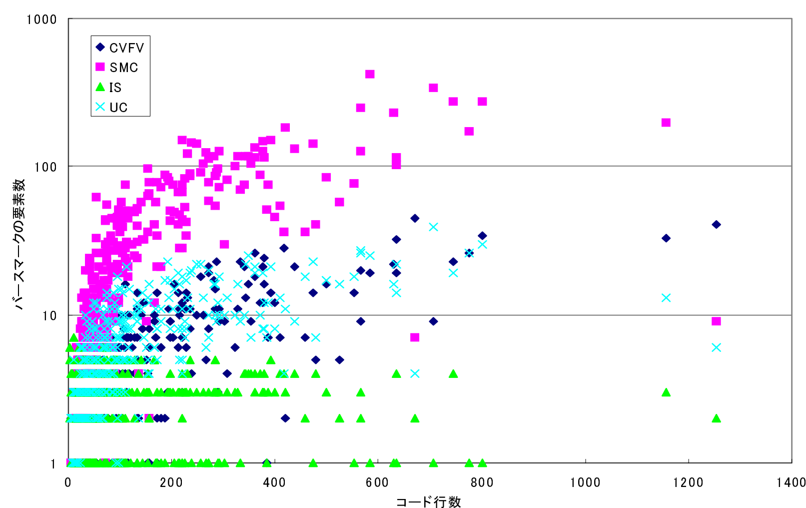

図 A.1 Apache Ant から抽出したバースマークの要素数の分布

要素数の少ないバースマークの要素の分布

図 A.1〜A.4 のグラフにおけるバースマークの要素数 0 から 30 までの範囲を図 A.5〜A.8 に示す．各グラフの横軸はコメント行，空行を除いたソースコードの行数を表し，縦軸は各バースマークの要素数を線形スケールで表す．

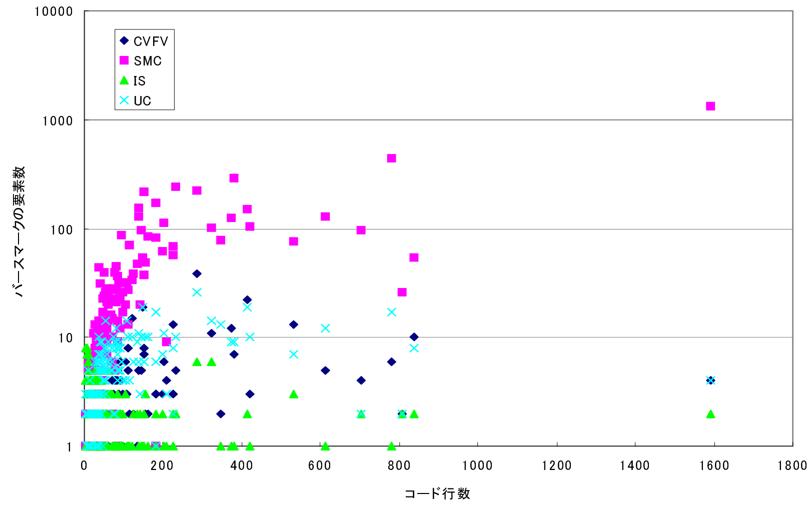

図 A.2 Jakarta BCEL から抽出したバースマークの要素数の分布

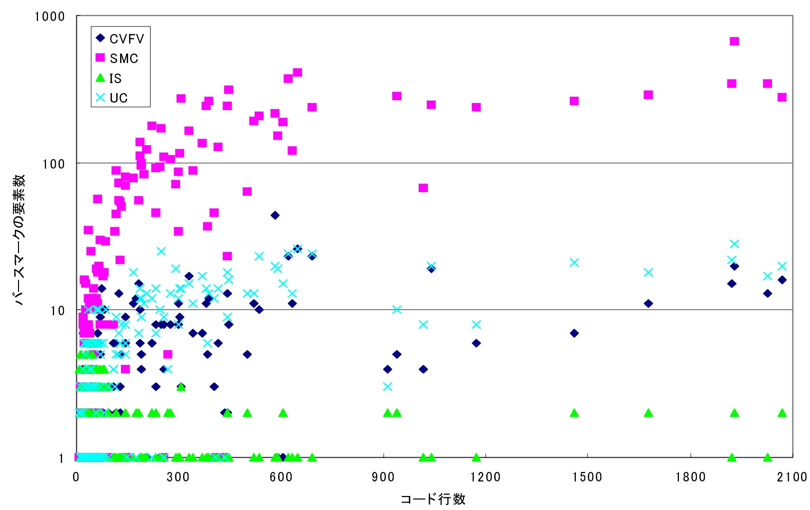

図 A.3 bloat から抽出したバースマークの要素数の分布

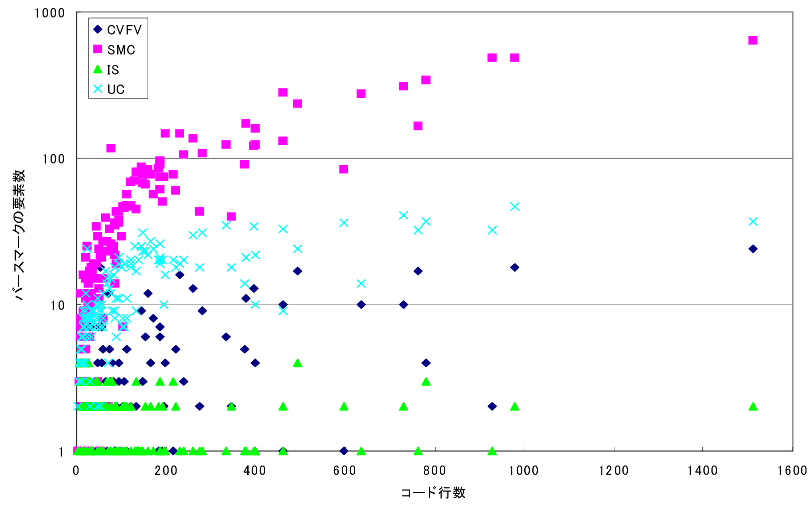

図 A.4 javassist から抽出したバースマークの要素数の分布

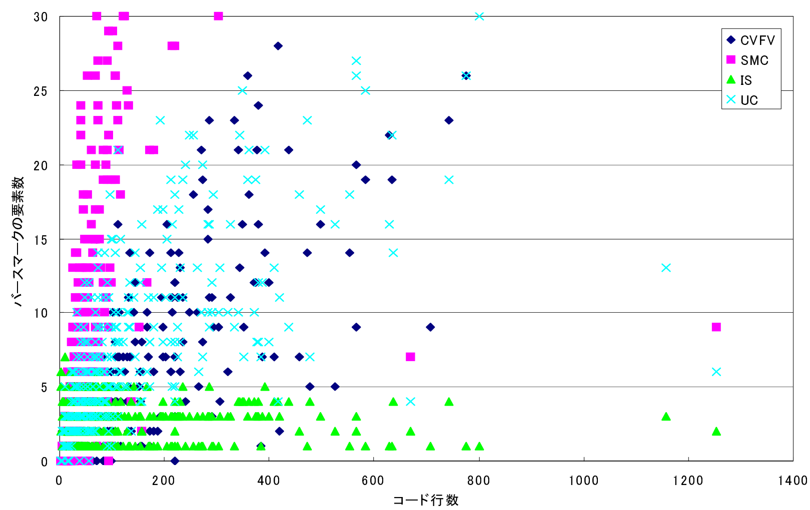

図 A.5 Apache Ant から抽出したバースマークの要素数の分布（0〜30）

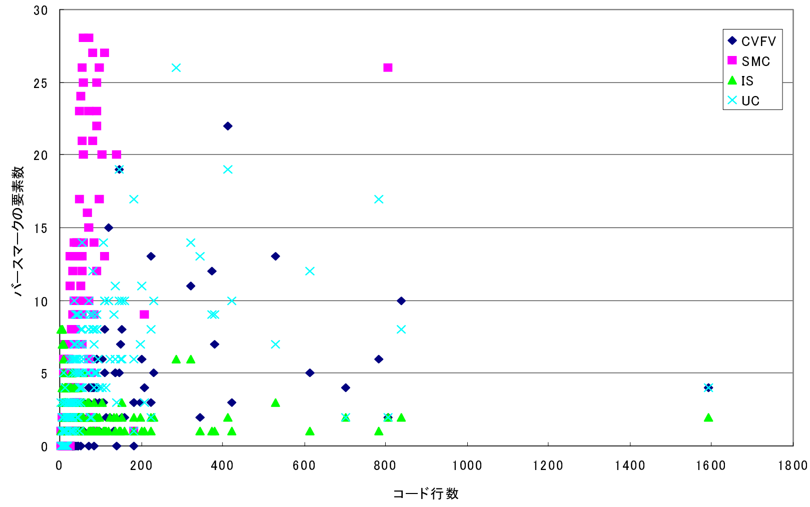

図 A.6　Jakarta BCEL から抽出したバースマークの要素数の分布（0〜30）

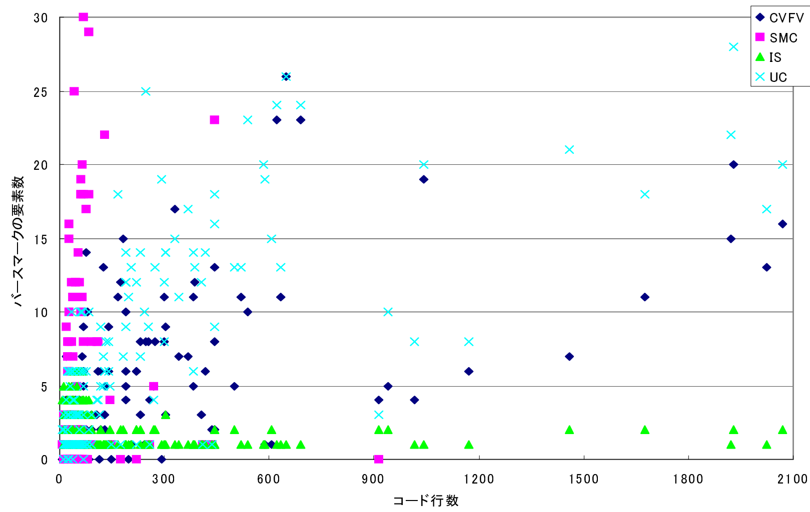

図 A.7　bloat から抽出したバースマークの要素数の分布（0〜30）

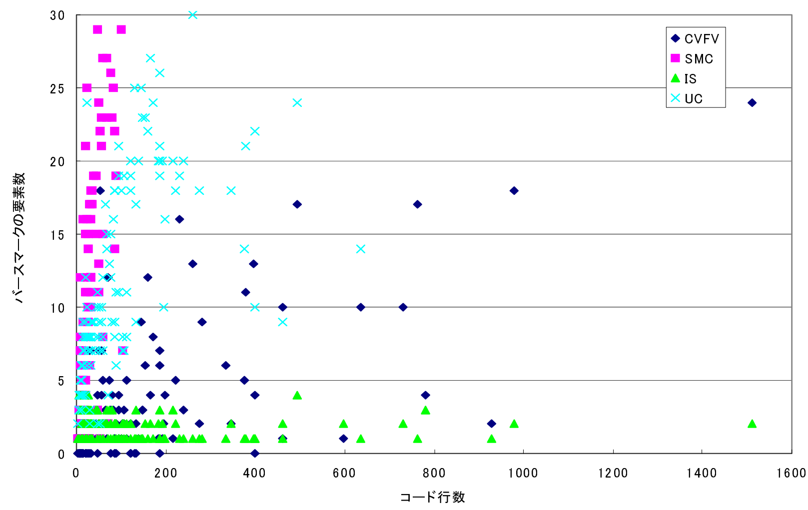

図 A.8 javassist から抽出したバースマークの要素数の分布（0〜30）

A.2 バースマークの要素数ごとのクラス数の分布

図 A.9〜A.12 にバースマークの要素数ごとのクラスの頻度を表すグラフを示す．横軸はバースマークの要素数を表し，縦軸はその要素数に占めるクラスの割合を表す．

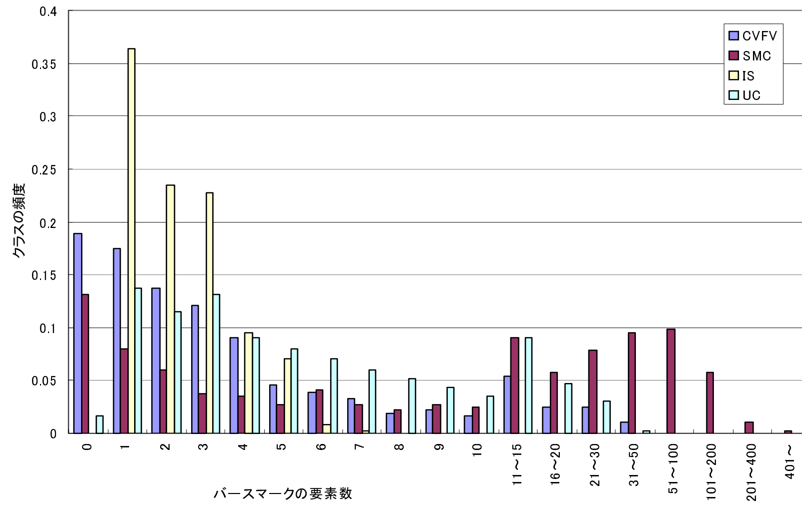

図 A.9 Apache Ant から抽出したバースマークの要素数の頻度

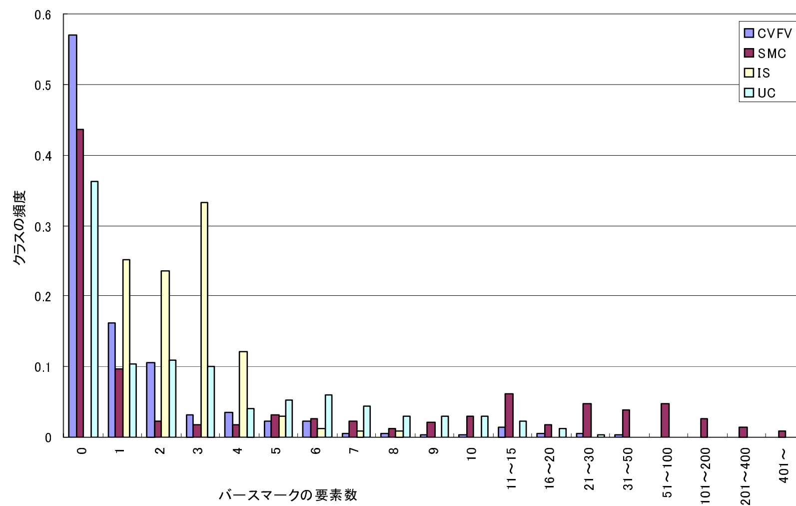

図 A.10 Jakarta BCEL から抽出したバースマークの要素数の頻度

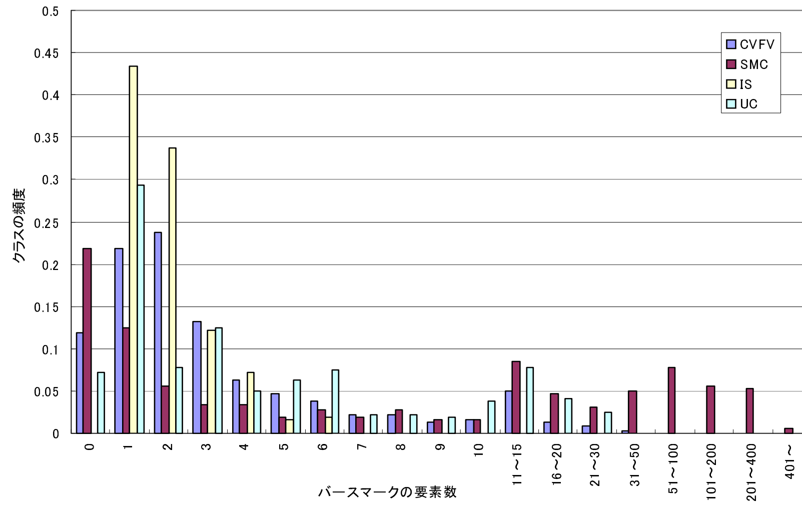

図 A.11 bloat から抽出したバースマークの要素数の頻度

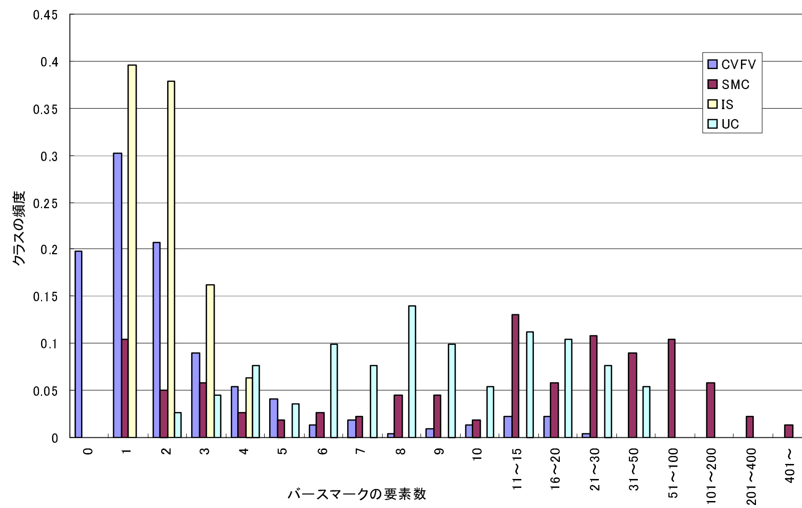

図 A.12 javassist から抽出したバースマークの要素数の頻度

B. 難読化例

典型的な Java プログラムを第 3 章で述べた提案難読化手法で難読化した例を示す．それぞれ，暗号アルゴリズムは DES を用い，鍵長は 56 ビットである．

B.1 Hello World

```
 1: public class HelloWorld{
 2:   public static void main(String[] args){
 3:     System.out.println("Hello World!");
 4:   }
 5: }
```

図 B.13 HelloWorld.java (オリジナル)

```
 1: public class HelloWorld{
 2:   public static void main(Object args){
 3:     DynamicCaller.invoke(
 4:       DynamicCaller.getStatic(
 5:         "965bc82af5b8182346eba0c13fc7074462d984001aa1a59f",
 6:         "c2036b9bc7a80b3a"
 7:       ),
 8:       new Object[] { "Hello World!" },
 9:       "d368580490143d48a523cc065b902eba3e74a1e1dba0e62f",
10:       "95b310aa7930cdf1"
11:     );
12:   }
13: }
```

図 B.14 HelloWorld.java (難読化後)

B.2 Fibonacci

```
 1: public class Fibonacci{
 2:   public int fibonacci(int n){
 3:     if(n <= 0)      throw new IllegalArgumentException();
 4:     else if(n <= 2) return 1;
 5:     return fibonacci(n - 1) + fibonacci(n - 2);
 6:   }
 7:   public static void main(String[] args){
 8:     Fibonacci f = new Fibonacci();
 9:     if(args.length == 0) f.fibonacci(3);
10:     else                 f.fibonacci(Integer.parseInt(args[0]));
11:   }
12: }
```

図 B.15 Fibonacci.java (オリジナル)

```
 1: public class Fibonacci{
 2:   public int fibonacci(int n){
 3:     if(n <= 0)
 4:       throw (Throwable)DynamicCaller.newInstance(new Object[0],
 5:         "d320f9973702ca278736cd9b164ffe30d18096c3203fd2de579e" +
 6:         "02e560823558a74d761386fa0ec6");
 7:     else if(n <= 2) return 1;
 8:     return ((Integer)DynamicCaller.invoke(
 9:         this, new Object[] { new Integer(n - 1) },
10:         "1b49a8d0c599bc7a7d8323b9af56f08c",
11:         "d1a084abc737263b7d8323b9af56f08c")).intValue() +
12:       ((Integer)DynamicCaller.invoke(
13:         this, new Object[] { new Integer(n - 2) },
14:         "1b49a8d0c599bc7a7d8323b9af56f08c",
15:         "d1a084abc737263b7d8323b9af56f08c")).intValue();
16:   }
17:   public static void main(String[] args){
18:     Object f = DynamicCaller.newInstance(new Object[0],
19:       "1b49a8d0c599bc7a7d8323b9af56f08c");
20:     if(args.length == 0) DynamicCaller.invoke(f, new Object[]
21:         { new Integer(3) }, "1b49a8d0c599bc7a7d8323b9af56f08c",
22:         "d1a084abc737263b7d8323b9af56f08c");
23:     else
24:       DynamicCaller.invoke(f, new Object[] {
25:           DynamicCaller.invokeStatic(new Object[] { args[0] },
26:             "d320f9973702ca27096a4efa7c27020c393d1dce360c8279",
27:             "9d4b21763742c4cc66b6556afca854d0") },
28:         "1b49a8d0c599bc7a7d8323b9af56f08c",
29:         "d1a084abc737263b7d8323b9af56f08c");
30:   }
31: }
```

図 B.16 Fibonacci.java (難読化後)