NAIST-IS-DT9661030

博士論文

知的財産権の保護を目的とする安全なソフトウェア実装法

門田 暁人

1998年8月28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。

門田 暁人

審査委員： 鳥居 宏次 教授
小山 正樹 教授
関 浩之 教授
松本 健一 助教授

知的財産権の保護を目的とする
安全なソフトウェア実装法*

門田 暁人

内容梗概

本論文では，ソフトウェア開発者の知的財産権を侵害する海賊行為の抑止を目的とした，ソフトウェアの安全な実装法を提案する．提案する方法は，安全な鍵付きプログラム実装法，プログラムコードを対象とした電子透かし挿入法，及び，プログラムの難読化法，の3方法である．これら三つの方法を目的に応じて使い分けたり併用することにより，ソフトウェアを無断でコピー，使用するといった比較的単純な海賊行為から，ソフトウェアを解析し含まれるアイデアを盗用するといった高度な海賊行為に至るまで，効果的に抑止することが出来る．本論文では，まず，第1章において，ソフトウェアの流通と海賊行為の現状を中心に当該分野の背景を述べ，解決すべき問題を整理する．次に，第2章では，従来の鍵付きプログラムを改良し，ネットワーク環境に対応した安全な鍵付きプログラムの実装法について述べる．第3章では，C，C++，JAVAなどのプログラムの実行ファイルを配布する際に，開発者の署名などの任意の文字列を電子透かしとして挿入，及び，取り出す方法を提案する．第4章では，人間が読んだり解析することが困難になるようにプログラムを難読化する方法を提案し，その有効性を実験によって示す．最後に第5章では，本論文の全体のまとめと考察を行う．

キーワード

ソフトウェア保護, 知的財産権, 鍵付きプログラム, 電子透かし, プログラムの難読化

---

*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-DT9661030, 1998年8月28日.

Protecting Intellectual Properties from Software Piracy*

Akito Monden

Abstract

This thesis proposes three methods for protecting software developers' intellectual properties: a method for incorporating a secured key into a program, a method for watermarking a program, and methods for scrambling a program. These three methods can be used to prevent a wide range of software piracy from very low level software piracy such as copying and using someone else's software with no permission, to high level piracy such as analyzing someone else's software and extracting ideas from it. Chapter 1 describes the status quo of software piracy, and clarifies the problems in protecting software developers' intellectual properties. Chapter 2 proposes a method for incorporating a secured key into a program that is applicable to network environments. Chapter 3 proposes a method for watermarking a program with software developers' signatures and/or software users' identifications. This watermarking method can be used for compiler languages such as C, C++, JAVA. Chapter 4 proposes methods for scrambling a program, which will result in a program that is difficult to analyze or understand. Chapter 5 concludes this thesis with a summary and future topics.

Keywords:

software protection, intellectual property, keyed program, watermark, program scrambling

*Doctor's Thesis, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT9661030, August 28, 1998.

関連発表論文

1. 門田暁人, 高田義広, 鳥居宏次, “プログラムの難読化法の提案,” 第51回情報処理学会全国大会, 5G-7, pp.4-263–4-264, Sep. 1995.

2. 門田暁人, 高田義広, 鳥居宏次, “プログラムの難読化法の実験的評価,” 情報処理学会研究報告, ソフトウェア工学研究会, 96-SE-108-5, pp.33–40, Mar. 1996.

3. 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.

4. 門田暁人, “コピー防止を目的とするプログラムの虫食い実装方式,” 情報処理学会 夏のプログラミングシンポジウム報告集, pp.47–54, July 1997.

5. 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'98-9.2.A, Jan. 1998.

6. Akito Monden, “A secure keyed program in network environment,” Proceedings of the 20th International Conference on Software Engineering, Vol.2, pp.170–171, Apr. 1998.

7. 門田暁人, “電子透かしによる複製防止技術,” 第2回コンピュータ犯罪に関する白浜シンポジウム, May 1998.

8. 門田暁人, 飯田元, 松本健一, 鳥居宏次, “安全な鍵付きプログラムによるソフトウェア保護,” ソフトウェアシンポジウム'98 論文集, pp.108–115, June 1998.

9. 門田暁人, 飯田元, 松本健一, 鳥居宏次, “ソフトウェアの不正利用の防止を目的とする安全な鍵付きプログラム,” 電子情報通信学会論文誌 D-I (投稿中).

10. Akito Monden, Hajimu Iida, Ken-ichi Matsumoto, Koji Torii, “An implementation of a secure keyed program in network environments,” submitted to International Journal of Software Engineering and Knowledge Engineering.

目 次

1. はじめに **1**

2. 安全な鍵付きプログラム **7**

2.1 あらまし 7
2.2 従来の鍵付きプログラムとその問題点 9
2.3 安全な鍵付きプログラム 13
2.3.1 安全な鍵付きプログラムの原理 13
2.3.2 安全な鍵付きプログラムの実装例 16
2.4 提案方法の安全性 18
2.4.1 従来の不正利用に対する安全性 18
2.4.2 予想される不正利用に対する安全性 19
2.5 Pay per use への応用 21
2.5.1 Pay per use の概念 21
2.5.2 安全な KP による pay per use の実現 22
2.6 まとめと今後の課題 23

3. プログラムに対する電子透かし法 **25**

3.1 あらまし 25
3.2 電子透かしが有用となる場合 26
3.2.1 違法コピーの抑止 26
3.2.2 改変・盗用プログラムの発見 26
3.2.3 プログラム改変・盗用の事実の証明 27
3.3 プログラムの電子透かしに要求される性質 28
3.4 電子透かしを挿入する方法 29
3.4.1 透かし挿入部の追加 31
3.4.2 目印の追加 32
3.4.3 透かしの埋め込み 33
3.4.4 透かしの取り出し 35

3.5 JAVAプログラムに対する電子透かし法 36
3.5.1 透かし挿入部の追加 37
3.5.2 透かしの埋め込み 37
3.5.3 プログラム変換に対する対策 38
3.5.4 透かしの挿入例 39
3.6 関連する研究 40
3.7 まとめと今後の課題 41

4. プログラムの難読化法 **43**
4.1 あらまし 43
4.2 難読化が有用となる場合 44
4.2.1 知的財産権を保護したい場合 44
4.2.2 システムに対する不正を防止したい場合 45
4.3 難読化についての定義 45
4.3.1 プログラムの解析 45
4.3.2 難読化の定義 46
4.4 ループを含むプログラムの難読化 47
4.5 提案する2通りの難読化の方法 48
4.5.1 制御構造を複雑化する方法 48
4.5.2 処理の順序を複雑化する方法 54
4.6 方法の評価実験 59
4.6.1 実験の方法 59
4.6.2 実験の結果 61
4.7 まとめと今後の課題 62

5. おわりに **63**

謝辞 **66**

参考文献 **67**

付録 72

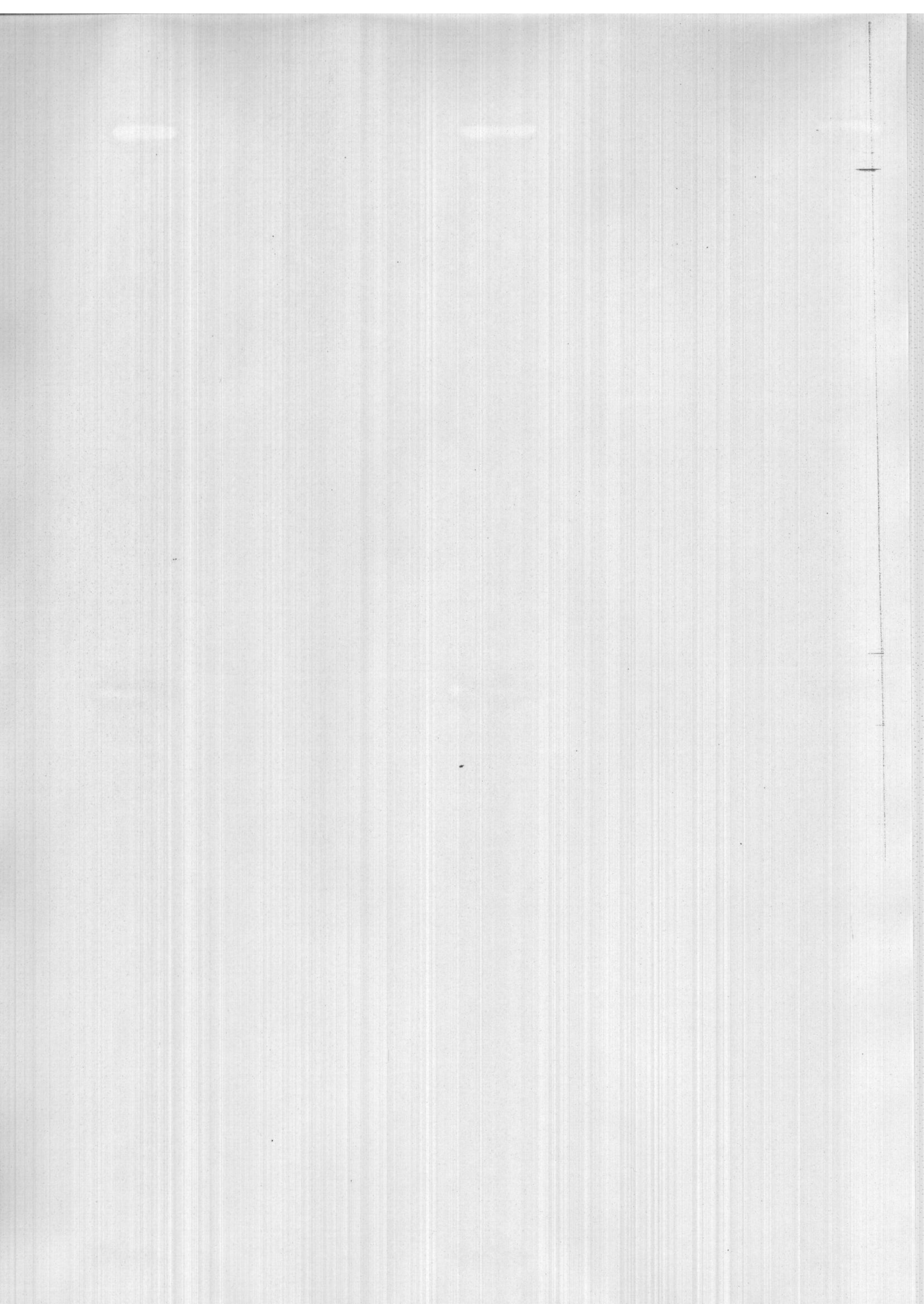
図目次

1 知的財産権の侵害となる海賊行為 2
2 海賊行為の分布 4
3 提案する方法の影響範囲 5
4 KP によるライセンス制御 8
5 安全な KP の概略 9
6 KP の動作原理 10
7 問題 1: 鍵の二重使用 11
8 問題 2: プログラムの改変 11
9 鍵管理システムによるライセンス制御 12
10 安全な KP の動作原理 13
11 安全な KP の通信プロトコル 15
12 安全な KP の実装例 17
13 問題 P_2（プログラムの改変）の解決 20
14 プロキシサーバによる不正 21
15 アドレス登録に対する不正 22
16 CGI プログラムによる安全な KP 24
17 提案する電子透かし法の概略 30
18 “assert” から “if” への変換 31
19 目印の必要性 32
20 目印の例 33
21 透かしの埋め込み 34
22 80x86/Pentium のバイナリプログラム例 35
23 電子透かし取り出しの問題 36
24 数値オペランドの取り出し 36
25 透かしを埋め込んだバイトコードの例 39
26 安全弁による補正を考慮した透かしの例 40
27 追加するメソッドの例 40
28 プログラムの難読化 43

29 ループを含むプログラム 47
30 フローチャート 49
31 規則に基づく変換 52
32 制御構造を複雑化したプログラム 53
33 処理の順序を複雑化したプログラム 57
34 処理の順序と制御構造を複雑化したプログラム 58

表目次

1 命令コードに対する情報の割り当て例 38
2 制御構造の変換規則 50
3 処理の順序の変換パターン 56
4 解析に要した時間と解析に失敗した回数 61



1. はじめに

多額の費用，労力，および高度な技術を投入して開発したソフトウェア（コンピュータプログラム）は，開発者の知的財産権が阻害されないように保護する必要がある．しかし，実際には，ソフトウェア開発者は知的財産権の侵害に常に脅かされているのが現状である．例えば，ソフトウェアの違法コピーは，長年に渡ってソフトウェア業界に莫大な損害を与え続けている [44][45]．インターネットの発展に伴って，海賊版ソフトウェアの流通も問題となっている [41]．ソフトウェアの解析によってソフトウェアに含まれるアイデアやアルゴリズム等が盗用されることも議論の対象となっている [38]．

本論文では，それらの知的財産権の侵害となる行為を「海賊行為」と呼ぶ．図1に，代表的な四つの海賊行為 p_1〜p_4を示す．なお，本論文の以降では，ソフトウェアのことを場合によりプログラムとも記述する．

(p_1) ソフトウェアのコピー．

フロッピーディスクやハードディスク，CD-ROM 等の記憶媒体に記録されているソフトウェアの複製を作成する行為である．同種類の媒体への複製はもちろん，異なる媒体への複製も含む．計算機のオペレーティングシステム上のコピーコマンドや，市販のコピーツール（バックアップソフトウェア）などによって行われる．

著作権法においては，ソフトウェアのコピー自体は必ずしも違法とならないが，コピーしたソフトウェアを著作権者の許諾なしに他人に配布したり販売すると著作権の侵害となる [8]．ソフトウェアの知的財産権の保護を目的とする代表的な非営利団体として知られている BSA（Business Software Alliance）[44] と SPA（Software Publishers Association）[45] の報告によると，1996 年に世界中で使用されたソフトウェアの 43%は違法コピーであり，違法コピーによるソフトウェア業界全体の推定損失額は 11.2 億ドルに達する．ソフトウェアの違法コピーの防止は，ソフトウェア業界における長年の重要な課題である．

(p_2) ソフトウェアの改変．

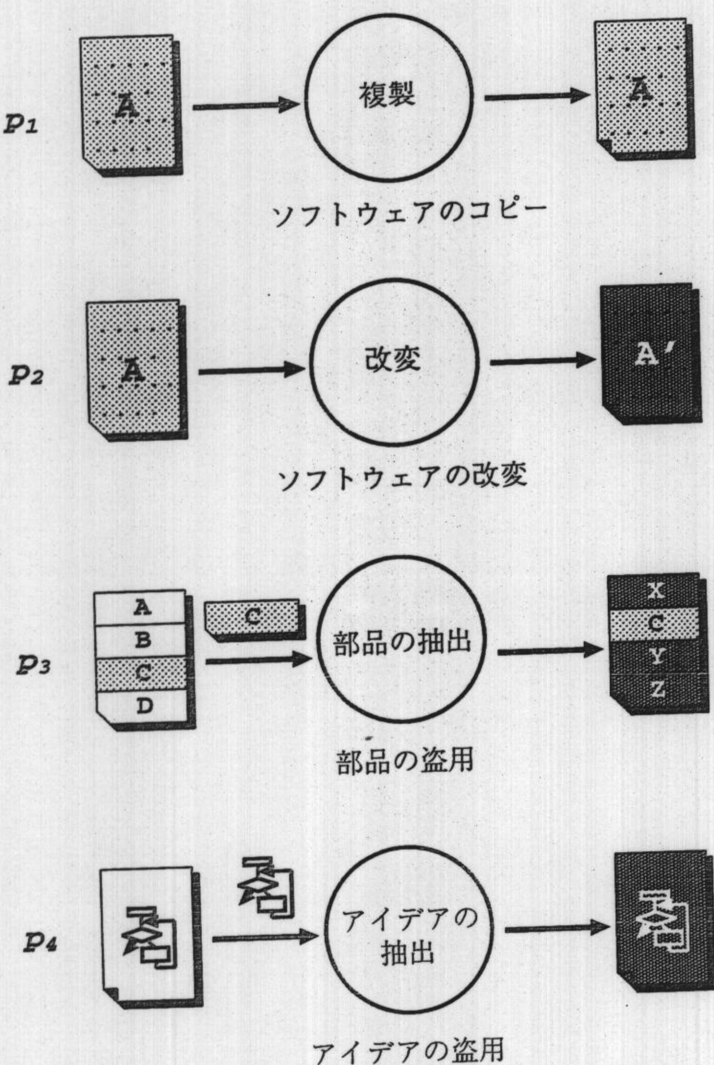


図 1 知的財産権の侵害となる海賊行為

他人の開発したソフトウェアを改造する行為である．本論文では，改変により得られたソフトウェアのことを，改変ソフトウェア（または，改変プログラム）と呼ぶ．

改変ソフトウェアは，コピーしたソフトウェアと同様，無断で配布，販売すると，著作権の侵害となる．ただし，改変の度合が大きくなればなるほど，改変したものであるのか新規に開発したものであるのかの客観的な区別が困難となり，改変の事実を立証することが困難となる．

(p_3) 部品の盗用．

他人の開発したソフトウェアの一部分を抽出し，自分のソフトウェアに組み込む行為である．本論文では，盗用した部品を組み込んだソフトウェアのことを，盗用ソフトウェア（または，盗用プログラム）と呼ぶ．

盗用ソフトウェアも，無断で配布，販売すると，著作権の侵害となる．ただし，実際には，ソフトウェアのごく一部分に盗用した部品が使われていても，それを発見することは容易ではない．また，ソフトウェアの改変（p_2）を組合せた海賊行為も考えられ，他人のソフトウェアの一部分を抽出し，抽出した部品を改変してから自分のソフトウェアに組み込んだ場合には，著作権侵害の立証が著しく困難となる．近年，JAVA や Active X といった再利用が容易なソフトウェア部品がインターネット上を流通することが増えてきており，それらのソフトウェア部品の盗用を防ぐことは，今後ますます重要になると考えられる．

(p_4) アイデアの盗用．

ソフトウェアを解析して，ソフトウェアに含まれるアルゴリズムなどのアイデアや方式を抽出し，他のソフトウェアに使用する行為である．

近年，アルゴリズムは特許による保護の対象となりつつあるが，ソフトウェア中のアイデアや方式がすべて特許の対象となるわけではない．また，ソフトウェア中のアイデアや方式が盗用されたとしても，それを立証することは容易でない．

上記の p_1 ～ p_4 の海賊行為は，いずれも，ある元となるソフトウェアから別のソフトウェアを新たに得る行為である．それぞれの海賊行為は，元となるソフトウェアと新たに得たソフトウェアとの表現上の類似度に差がある．この類似度に基づいて各海賊行為を分類すると，図2のようになる．なお，図の横軸は，各海賊行為を行うのに必要となる「ソフトウェアの理解度」にもほぼ一致する．

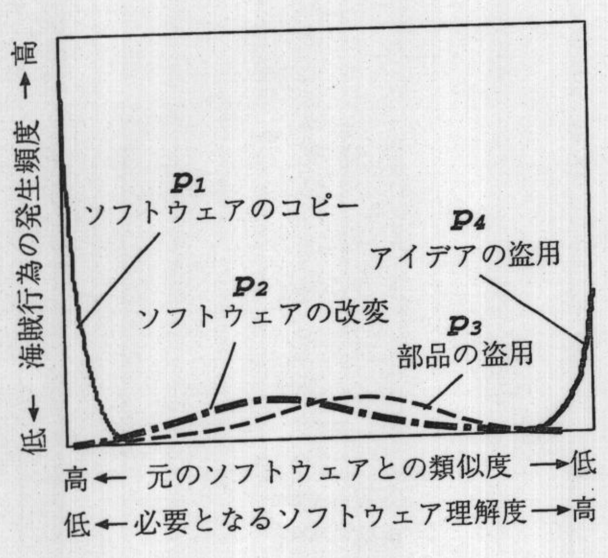


図 2 海賊行為の分布

元となるソフトウェアと新たに得たソフトウェアの類似度から p_1～p_4を分類すると，p_1のソフトウェアのコピーは，類似度がもっとも高い海賊行為である．特に，いわゆるデッドコピーでは，類似度は最大となる．p_2のソフトウェアの改変は，類似度がやや小さいコピーに当たる．ただし，類似度が著しく小さい場合には，もはやソフトウェアの改変とは呼べず，新規ソフトウェアの開発とみなすことができる．p_3の部品の盗用では，新たに得たソフトウェアは，盗用した部分を除いては元となるソフトウェアと類似しておらず，類似度は小さい場合が多いと考えられる．類似度が極めて大きい，すなわち，ソフトウェアの大部分を盗用した場合には，ソフトウェアのコピーや改変の範疇に含まれる．p_4のアイデアの盗用においては，元となるソフトウェアから抽出したアイデアのみを元に新たなソフトウェアを作成するため，ソフトウェアの表現上の類似度は極めて小さい．類

似度が大きい場合には，ソフトウェアの改変や部品の盗用の範疇に含まれると考えられる．

一方，各海賊行為を行うのに必要となるソフトウェア理解度の面から p_1〜p_4を分類すると，ソフトウェアのコピー（p_1）を行うには，対象となるソフトウェアをさほど理解する必要はなく，必要となる理解度は小さい．ソフトウェアの改変（p_2）や部品の盗用（p_3）を行うには，ソフトウェアをある程度理解する必要がある．アイデアの盗用（p_4）を行うには，ソフトウェアに対するさらに深い知識が必要となる．

本論文では，p_1〜p_4 の海賊行為を抑止を目的とする，三つのソフトウェア実装法（安全な鍵付きプログラム，プログラムを対象とする電子透かし法，プログラムの難読化法）を提案する．それぞれの実装法は，図3に示す部分の海賊行為の抑止に役立つ．

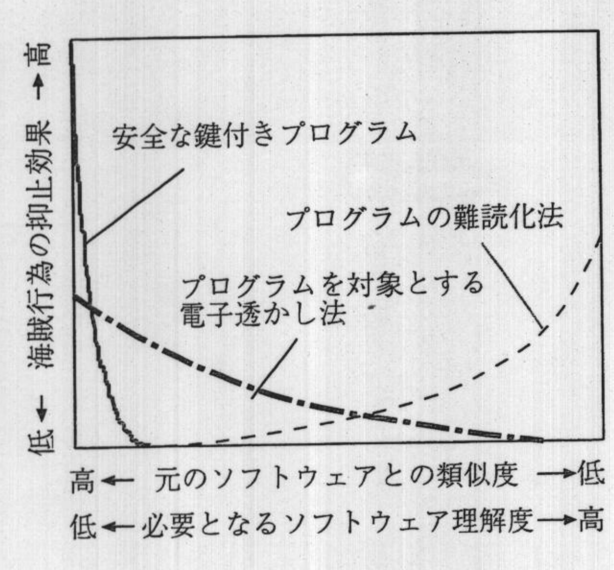


図 3 提案する方法の影響範囲

図3は，本論文で提案する三つのソフトウェア実装法の位置付けを表している．図から分かるように，三つの方法は，互いを補う形で，ソフトウェアに対する海賊行為を抑止する効果がある．安全な鍵付きプログラムは，ソフトウェアのコピーを抑止し，その抑止効果は大きい．プログラムを対象とする電子透かし法は，ソ

フトウェアのコピー，改変，盗用を抑止する．プログラムの難読化は，主にアイデアの盗用を抑止する．

本論文では，まず，2章において，ネットワーク環境における安全な鍵付きプログラムを提案する．鍵付きプログラムとは，鍵を持つ正規ユーザだけに実行が許されるプログラムのことである．鍵付きプログラムがコピーされたとしても，鍵を持たない不正ユーザによるプログラムの実行が防止されるため，結果的にプログラムのコピーが抑止される．提案する方法では，従来の鍵付きプログラムでは防ぐことが困難であった (1) 複数ユーザが同じ鍵を同時に使用する，(2) 鍵がなくても動作するようにプログラムを改変する，といった不正利用を防止することが可能となる．

3章では，プログラムの実行ファイル（バイナリプログラム）に，任意の文字列を電子透かしとして挿入する方法を提案する．ソフトウェアをユーザに配布する前に，ソフトウェア開発者の署名や各ユーザの識別子等を電子透かしとして挿入しておくことで，海賊行為を行った者を特定したり，海賊行為が行われたという事実を立証するのに役立つ．ただし，ソフトウェアの類似度が小さくなるような海賊行為では，ソフトウェア中の電子透かしが消えてしまう恐れがあるため，抑止効果は小さくなる．

4章では，プログラムを難読化する方法を提案する．プログラムの難読化とは，人間がプログラムを理解したり解析することが困難になるように，与えられたプログラムを著しく読みにくいプログラムへと変換する技術である．プログラムを難読化してからユーザに配布することで，ユーザや第三者によるプログラムの解析を困難にできるようになる．プログラムの解析を困難にすることは，アイデアの盗用を防ぐだけでなく，ソフトウェアの改変や部品の盗用の抑止にも役立つ．ソフトウェアを改変したり部品を抽出する場合にも，ソフトウェアを解析して理解する必要があるためである．

最後に，5章では，本論文の全体のまとめと考察を行う．

2. 安全な鍵付きプログラム

2.1 あらまし

ソフトウェア（コンピュータプログラム）の違法コピーが，ソフトウェア開発者の知的財産権を守る上で大きな問題となっている [7][33]．例えば，他人が所有する市販ソフトウェアをコピーして利用する，シングルユーザライセンスしか与えられていないプログラムを会社組織内の複数の計算機にインストールして利用する，といった行為は違法コピーにあたる．実際，これらの違法コピーは，多くの計算機環境において容易に行われ得る状況にあり，ソフトウェア業界に大きな損害を与えている [44][45].

ソフトウェアの違法コピーを防止するために，従来より，鍵付きプログラム（Keyed Program，以下 KP と記す）が開発されてきた [9][16][36]．KP は，図 4に示すように，鍵を持つ正規ユーザ（User）だけに使用が許されるプログラムである．ここでいう鍵（Key）とは，特殊なフォーマットが施されたフロッピーディスク（Key FD），ハードウェアキー，プログラムのシリアル番号，プログラムマニュアルの記載事項などである．例えば，Microsoft Windows95 では，インストール時に *CD-key* と呼ばれる鍵（十数桁の数字）が必要となる．鍵を持たない不正ユーザによるプログラムの実行が防止されるため，結果的にプログラムのコピーが防止される.

従来の KP では，複数ユーザが同時に同じ鍵を使用する（鍵の二重使用），鍵がなくても動作するように KP の一部を書き換える（KP の改変），といった不正利用を防止することは困難である [29][30][32]．特に，KP の改変の問題は深刻である．ひとたびプログラムがユーザに配布されると，その改変を防ぐことは容易ではない.

本章では，従来の KP の問題を解決するための，ネットワーク環境における安全な KP を提案する．図 5に，提案する方法の概略を示す．従来の KP と異なり，ユーザには KP 全体は配布せず，KP の重要なごく一部分（KP fragment）を切りとってプログラム開発者（又は販売者）の管理下にある計算機（KP center）に置く．そして，切りとった残りの部分の KP（KP body）をユーザに配布する．ユー

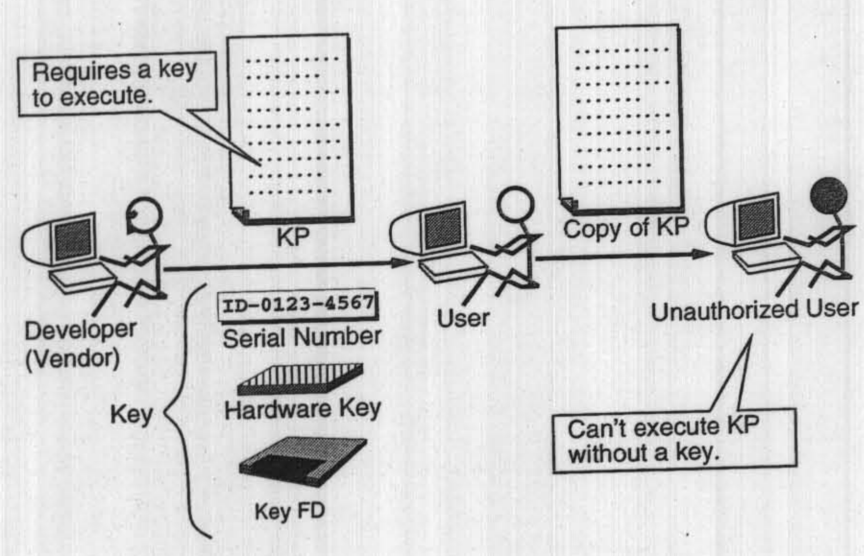


図 4 KP によるライセンス制御

ザは，一部分の欠けた不完全な KP しか持たないため，KP の実行時に KP center にアクセスして KP fragment の実行結果を受けとる必要がある．本方法では，ユーザに配布した KP 本体（KP body）ではなく，KP center 上の KP fragment に鍵を掛けておくことで，鍵の二重使用が防止できる．しかも，KP fragment がユーザから隠蔽されているため，KP center にアクセスしなくても動作するように KP を改変することは著しく困難であり，KP の改変も防止できる．なお，本方法においては，実行頻度の小さなごく一部分のプログラム（KP fragment）だけを KP center に置くため，KP center やネットワークにかかる負荷は小さく，多数のユーザに配布するようなプログラムに対しても適用できる．

提案する方法は，単にソフトウェアの違法コピーを防止するのみでなく，超流通 [33][40] に代表されるような新しいソフトウェア流通システムの実現に役立つ．超流通の基本となる考え方は，ソフトウェアを無料で配布する代わりにその使用頻度に応じてユーザに課金する（pay per use）ことで，ソフトウェアの流通を飛躍的に促進することである．pay per use においては，ユーザは，使いもしないプログラムに大金を払うというリスクから解放される．また，開発者にとっても，多くのユーザにプログラムを試用してもらえるという利点がある．提案方法では，

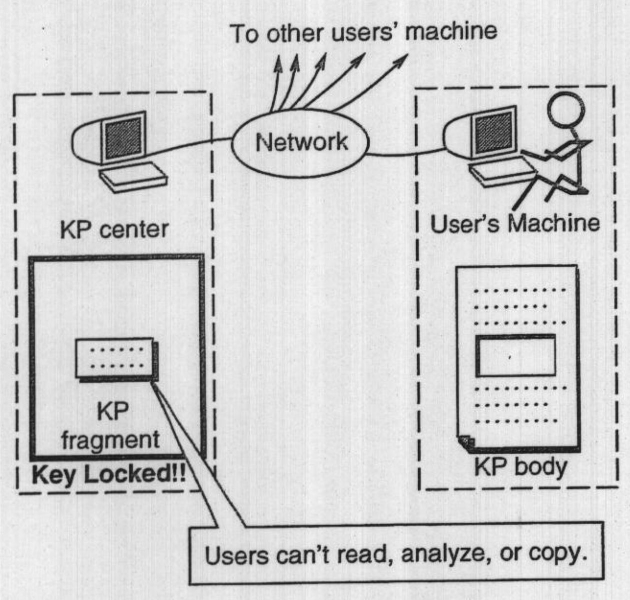


図 5 安全な KP の概略

ソフトウェアの使用履歴を KP center で記録できるため，特殊なハードウェアを用いることなく pay per use の仕組みを安全に実現できる．

本章の以降では，**2.2** において，従来の KP とその問題点を述べる．その問題点を解決する方法として，**2.3** では，安全な KP の動作原理と実装例を述べる．**2.4** では，提案する方法が **2.2** で述べた問題点を解決できることを示す．**2.5** では，提案方法による pay per use の実現について述べる．最後に，**2.6** では，まとめと課題を述べる．

2.2 従来の鍵付きプログラムとその問題点

プログラム開発者の知的財産権を保護する上では，プログラムの違法コピーを防止する，もしくは，違法コピーされたプログラムの実行を防止することが重要である．ただし，現状の計算機アーキテクチャにおいては，ユーザに配布したプログラムのコピーを防止することは困難である．そのため，コピーそのものは防止せずに，コピーされたプログラムの実行を防止する方法が，従来，数多く開発

されてきた．KP とは，そのような実行防止を目的として実装されるプログラムのことである．

KP の動作原理を図 6に示す．図に示すように，KP は，鍵の正当性を検査するためのプログラム（鍵のチェックルーチン）を含む．ユーザが KP に鍵を与えた時に，鍵のチェックルーチンが鍵の正当性を検査する．鍵が正当であった場合には，プログラム本体が実行される．KP の種類によっては，KP のインストール時のみでなく，KP の実行中にも鍵のチェックが行われる．

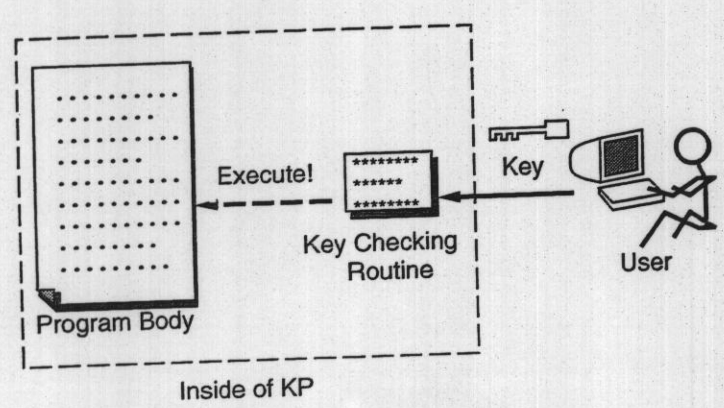


図 6 KP の動作原理

従来の KP には主に次の二つの実用上の問題がある．

[問題 P_1: 鍵の二重使用] KP がコピーされ，それらが 1 個の鍵によって複数ユーザに同時に使用される．

ライセンスを持たないユーザが，正規ユーザの鍵を使って不正にコピーした KP を実行する様子を図 7に示す．この場合，複数ユーザが同じ鍵を用いて同時に複数の KP を使用することになり，シングルユーザライセンスが守られていないことになる．

問題 P_1は，鍵を持たない者が，鍵の全数探索や KP の解析などにより有効な鍵を発見した場合にも発生する．例えば，プログラムの製品番号を鍵として用いた場合には，ユーザが KP のコピーさえ持っていれば，でたらめな鍵を数多く KP に入力することにより有効な鍵を発見できる可能性がある．また，鍵のチェック

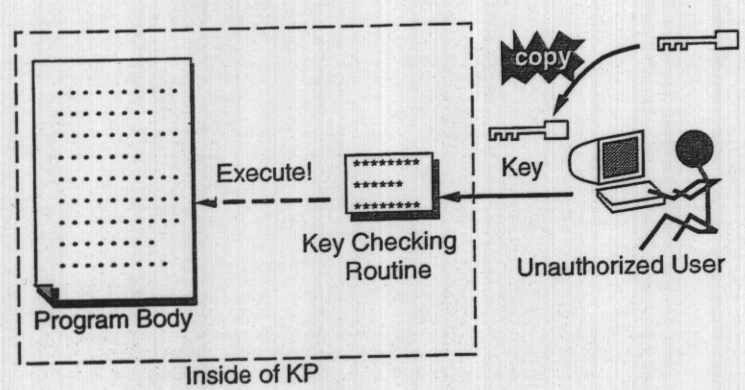


図 7 問題 1: 鍵の二重使用

ルーチンが解析されることにより，有効な鍵の集合が知られる可能性もある．プログラムを改変して鍵のチェックルーチンを壊すツールや，プログラムを解析するためのツールも現実に普及している [41].
[問題 P_2: プログラムの改変] 正当な鍵がなくても動作するようにプログラムが改変される（図 8参照）.

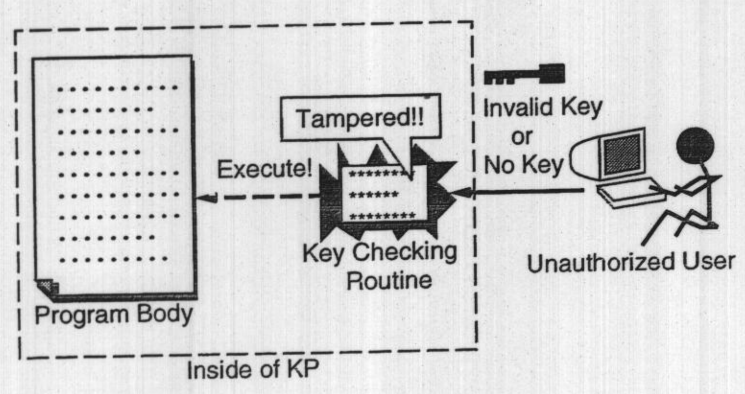


図 8 問題 2: プログラムの改変

フロッピーディスクがソフトウェア流通の主流を占めていた頃は，特殊フォーマットを施した Key FD を鍵として用いる KP が頻繁に開発されてきた [9]．しかし，通常のフォーマットを施したフロッピーディスクでも動作するように KP

を改変するツールもまた頻繁に流通していた．

従来，問題 P_1 や P_2 の解決を目的として，いくつかのシステムや方法が提案されている．例えば，ソフトウェアの同時使用ライセンスの管理を目的として，LAN や WAN 上で KP の鍵管理を行うシステムが提案されている．同時使用ライセンスとは，同時に使用できるソフトウェアの数を制限するライセンス契約である．図 9に，鍵管理システムの一例を示す [16]．この方法では，鍵サーバが鍵の同時使用数を制御することにより，同時に起動できるソフトウェアの数を制限する．それぞれのクライアントの計算機上には，KP がインストールされる．クライアントが KP を実行する際には，鍵サーバに鍵の送信を依頼する．鍵サーバは KP の同時実行数を検査し，最大数に達していなければ，ネットワークを通して鍵をクライアント計算機に送信する．ただし，この方法は，企業の管理責任者が従業員による不正行為を防止する場合には役立つが，個人ユーザによる不正行為を必ずしも防止できない．ユーザの手元にプログラムがある以上は問題 P_2（プログラムの改変）を解決できないためである．

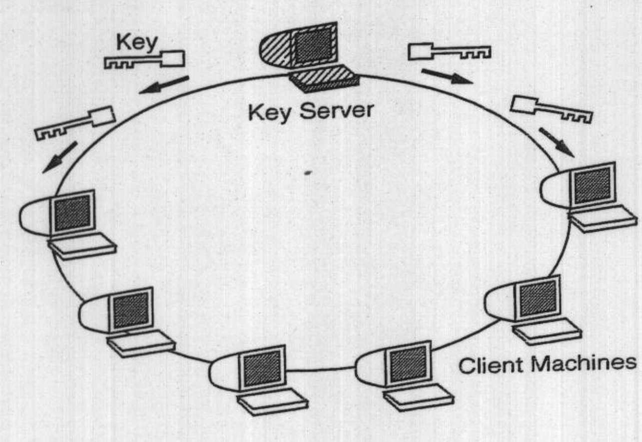


図 9 鍵管理システムによるライセンス制御

ライセンスを持たないユーザによるプログラムの不正利用を防止するために，暗号技術を用いる方法も提案されている [1][36]．それらの方法では，暗号化したプログラムと復号のための鍵がユーザに配布される．そのため，配布過程においてはプログラムの解析や改変が防止できる．しかし，実行時にはプログラムが復号

されるため，復号後のプログラムがコピー，解析，又は，改変される恐れがある．

復号後のプログラムの解析や改変を困難にするために，プログラムを分割してそれぞれを暗号化し，段階的に復号しながら実行する方法も提案されている[3][17]．しかし，既存のコンパイラを用いてこのようなプログラムを実装することは必ずしも容易でなく，実装に多大なコストがかかるという問題がある．また，復号後のプログラムのコピーや解析を困難にできたとしても，復号前のプログラムはコピー可能なため，問題 P_1（鍵の二重使用）は必ずしも解決されない．

2.3 安全な鍵付きプログラム

2.3.1 安全な鍵付きプログラムの原理

提案する方法の動作原理を図 10に示す．本方法では，KP fragment と鍵の検査ルーチンが KP から分離され，KP center に設置される．そして，残りの KP（KP body），および，KP center へのアクセス鍵が，ユーザに配布される．KP body 実行時に KP fragment の実行が必要になると，鍵がセンタに送信される．鍵が正当なものであるならば，KP fragment が実行され，実行結果が KP body へ返される．なお，本論文では，ユーザの計算機は，定常的にネットワークに接続されていると仮定し，いわゆるモバイルコンピュータは対象外とする．また，ユーザの計算機と KP center の間のネットワークは十分にロバストであると仮定する．

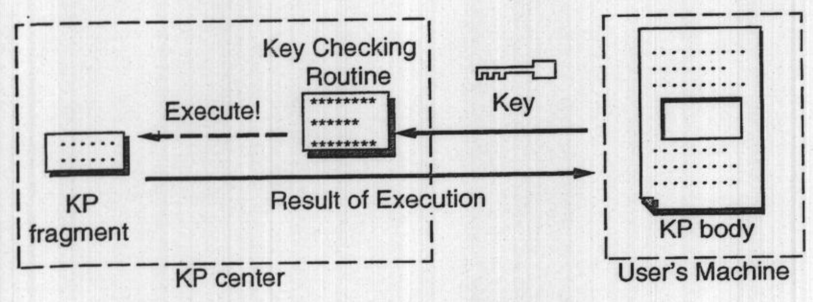


図 10 安全な KP の動作原理

ユーザの計算機と KP center 間の通信プロトコルを図11に示す．図中の記号の意味は，次の通りである．

KPbody: ユーザが購入したプログラム．

Key: ネットワークを通して KP center にアクセスするための鍵．ソフトウェアの製品番号のように，数値や文字列で表される．鍵には重複がないものとし，各ユーザへはそれぞれ異なる鍵が配布される．正当な鍵の集合は，予め KP center に登録されているものとする．

Adr: ユーザの計算機のネットワークアドレス．KP center においてユーザの計算機を区別するために用いられる．計算機毎に固有の値を持ち，変更されることもないとする．

Inp: KP fragment への入力変数．KP fragment が実行される度に異なる値をとり得る．

Out: KP fragment の出力値（実行結果）．*Inp* と同様，KP fragment が実行される度に異なる値をとり得る．

以下に，プロトコルの詳細を示す．

1. KP body の購入

   ユーザは，ソフトウェア（KP body）を購入する時に，KP center にアクセスするための鍵（*Key*）も同時に受けとる．

2. KP body のインストール

   ユーザは，鍵 *Key*をキーボードから入力することを要求される．KP body は，*Key*と *Adr*を KP center に送信する．KP center は，*Key*が正当なものであれば，*Key*に対して *Adr*の登録を行う．

3. KP body の実行

   KP body の欠損部（KP fragment を切りとった部分）にプログラムの実行が達した場合，*Inp*，*Key*，*Adr*が KP center へ送信される．KP center は，*Key*に対して *Adr*が登録されているかどうかを検査し，登録されているならば，*Inp* を入力として KP fragment を実行し，*Adr*で指定される計算機

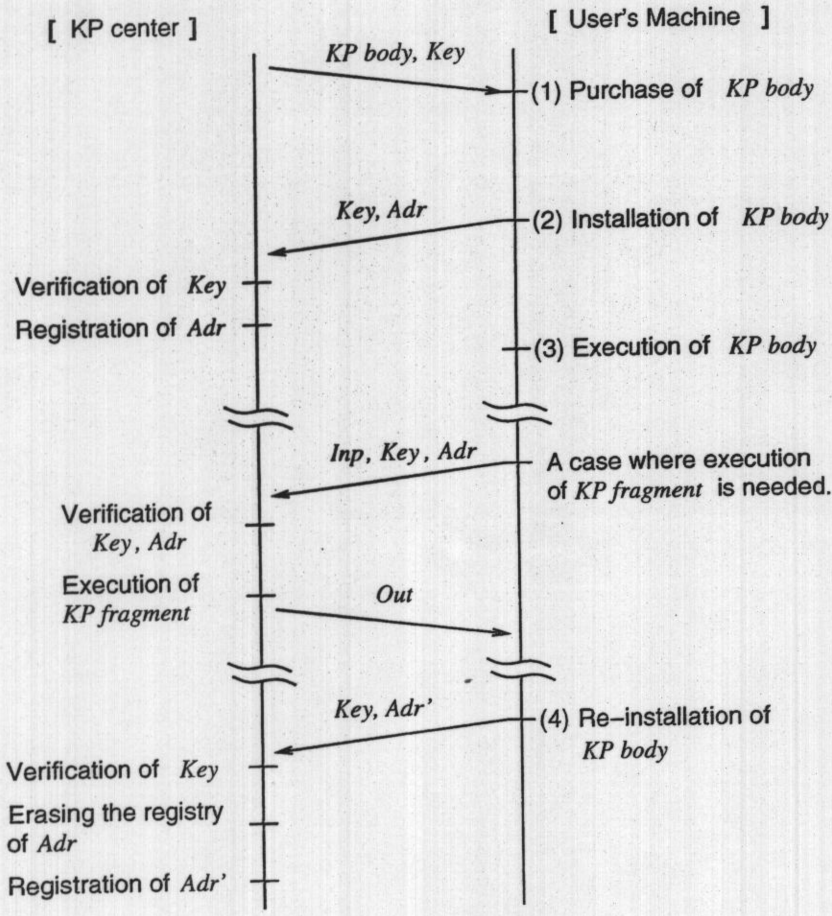


図 11 安全な KP の通信プロトコル

に *Out* を返す．*Adr*が未登録のものであるならば，KP center は *Out* を返さない．従って，KP body のインストール時に *Adr*を登録した計算機上でのみ，KP body の実行が許されることになる．

4. KP body の再インストール

   ユーザが，アドレス *Adr'*をもつ計算機で KP body を再インストールした場合，古いアドレス *Adr*の登録が抹消され，新しいアドレス *Adr'*が登録される．

2.3.2 安全な鍵付きプログラムの実装例

提案する方法の単純な（かつ部分的な）C 言語による実装例を図 12に示す．図 12(a) は，提案する方法を適用する前のプログラムを示し，図 12(b) は適用後のプログラムを示す．図中の関数 *Match* は，ストリングマッチングを行うプログラムである．この例では，本方法の適用によって，式 "$j == M$" が元のプログラムより切り取られ，関数 *KP_Frgm*（KP fragment）として KP center 上に設置される．切り取られた後の部分は，関数呼び出し "*Call_KP_Center(j, M);*" で置き換えられる．また，図中には明示されていないが，関数 *Call_KP_Center* は，変数 j，M（KP fragment への入力 *Inp* となる），*Key*，*Adr*を KP center へ送信し，*Out* を受信する．従って，関数 *Call_KP_Center* が呼び出されると KP center において *Key*と *Adr*が検査され，関数 *KP_Frgm* が実行されることになる．

提案する方法では，KP fragment とする部分を注意深く選定する必要がある．例えば，選定の基準としては次のようなものが考えられる．

- KP center とネットワークの負荷が大きくなりすぎない程度に，実行頻度が小さいかどうか
- KP center へのアクセスにおける待ち時間が短い，あるいは，見かけ上短いかどうか

KP fragment の候補としては，ファイルのロード，セーブ，あるいは，プリンタへの出力の際に実行されるコードが考えられる．ファイルのロード，セーブ，及

```
int
Match(int A[], int B[], int N, int M)
{
    int i, j;
    for (i = 0; i <= N - M; i++) {
        for ( j = 0; j < M; j++) {
            if(A[i + j] != B[j]) {
                break;
            }
        }
        if (j == M) {
            return i;
        }
    }
    return -1;
}
```

(a) Original String Matching Program

KP fragment (KP center):

```
.....

int
KP_Frgm(int j, int M)
{
    return j == M;
}

.....
```

KP body (User's Machine):

```
int
Match(int A[], int B[], int N, int M)
{
    int i, j;
    for (i = 0; i <= N - M; i++) {
        for ( j = 0; j < M; j++) {
            if(A[i + j] != B[j]) {
                break;
            }
        }
        if (Call_KP_Center(j , M)) {
            return i;
        }
    }
    return -1;
}
int
Call_KP_Center(int j, int M)
{
    ...
}
```

(b) Modified Program with Secure KP Implementation

図 12 安全な KP の実装例

び，プリンタへの出力の機能は，アプリケーションプログラムの多くが備えており，実行される頻度も比較的小さい．多くの場合，数分から数時間に1回の割合でしか実行されない．また，それらの機能の実行には，周辺機器との比較的低速なデータ通信を伴うため，もとよりある程度の待ち時間は避けられない．従って，KP center へのアクセスをデータ通信のバックグラウンドで行う等の工夫により，見かけ上の待ち時間を短くすることができる．

2.4 提案方法の安全性

2.4.1 従来の不正利用に対する安全性

問題 P_1（鍵の二重使用）の解決
KP center では，1個の鍵に対しては1個のネットワークアドレスしか登録できないので，鍵の二重使用は防止できる．また，鍵の検査ルーチンはユーザの手元にないので，鍵の全数探索や鍵の検査ルーチンの解析による鍵の発見も防止できる．例えば，不正ユーザが，鍵の全数探索のために，KP body のインストール時にでたらめな鍵を1個ずつ KP center に送信しているとする．この場合，KP center において，KP body のインストール時の鍵の検査にわざと遅延時間を設けておき（例えば30秒程度），かつ，鍵空間を大きくしておくことにより，鍵の探索を事実上不可能とすることができる．KP body のインストール時のみに遅延時間を設けることは，正規のユーザには大きな負担とならない．

なお，提案方法では，鍵の二重使用が防止できる代わりに，鍵の番号が他人に盗用されると正規ユーザがプログラムを実行できなくなるという問題がある．この問題は，KP body の販売時に予め予備の鍵を正規ユーザに渡しておくことによって解決できる．鍵が盗用されたと判断した正規ユーザが予備の鍵を使い始めた時点で，古い鍵の使用を自動的に停止すればよい．

鍵が盗用されると正規ユーザもプログラムを実行できなくなることは，一方で，正規ユーザが不正利用に関わることを抑止する効果がある．従来の KP では，プログラムを不正にコピーして他人に渡したり，他人に鍵番号を教えても，正規ユーザ自身が直接不利益を被ることはなく，不正利用を助長する面があった．提案方法では，不正利用によって不利益を被るのは正規ユーザ自身である．多くの正規

ユーザは，UNIXのログインパスワード等と同様の厳密さで鍵を保管すると思われる．

なお，提案方法は，複数ユーザライセンスにも適用できる．複数ユーザライセンスとは，1個のプログラムに対する使用許可を二人以上のユーザに与える許諾方法である．KPにおいて複数ユーザライセンスを実現するためには，1個の鍵を複数人で共有することを許せばよい．提案方法では，1個の鍵に対して複数のアドレスの登録を許せば，複数ユーザライセンスを与えることができる．

問題 P_2（KPの改変）の解決

提案方法では，次の三つの理由により，正当な鍵なしで動作するようにプログラムを改変することは非常に困難である（図13参照）．

1. 鍵のチェックルーチンは，KP centerにあるためユーザは改変できない．
2. KP bodyは改変可能であるが，どのように改変しようともKP centerへのアクセスは必要であり，鍵なしでKP bodyを実行することはできない．これは，KP fragmentの実行結果 Inp が，KP bodyの実行に必要不可欠なためである．
3. KP center上の鍵のチェックルーチンをバイパスしてKP fragmentに直接アクセスすることはできない．これは，KP fragmentはユーザから隠蔽されているためである．

なお，KP fragmentの仕様は，不正ユーザに容易に推測されない方が望ましい．KP fragmentの仕様が知られると，偽のKP fragmentを作成し，KP centerにアクセスすることなくKP bodyを動作させることが可能になる．KP fragmentの仕様を推測されにくくする一つの方法は，入力変数 Inp，及び，出力結果 Out にダミーのデータを含めることである．

2.4.2 予想される不正利用に対する安全性

プロキシサーバによる不正 KP centerへのアクセスを代行する計算機（プロキシサーバ）を用意すれば，多数の不正利用者が1個の鍵を共有することが可能に

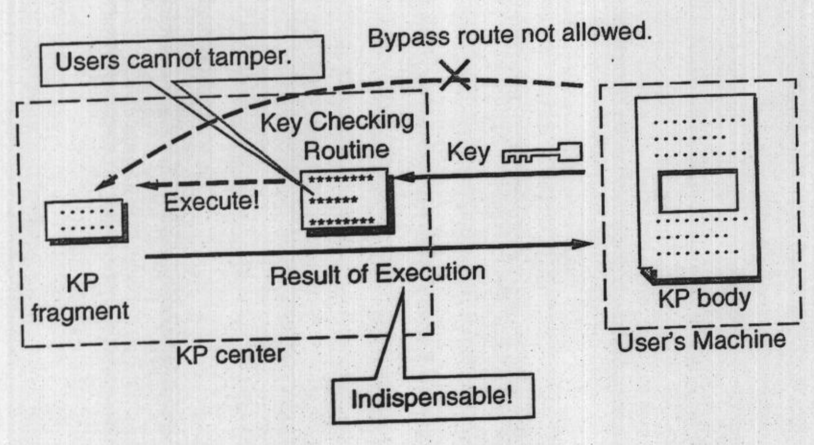


図 13 問題 P_2（プログラムの改変）の解決

なる（図 14参照）．不正利用者は，KP center に直接アクセスせずに，プロキシサーバに対して KP fragment への入力 *Inp* を送信し，プロキシサーバ経由で実行結果 *Out* を受け取る．プロキシサーバが鍵を 1 個だけ持っていれば，複数の人間が KP body を実行できることになる．

この不正利用を防止する一つの方法は，各アドレスからの KP center へのアクセス頻度を常にチェックすることである．もし，特定のアドレスからのアクセス頻度が極端に高ければ，そのアドレスからのアクセスを KP center が無視するといった処置をとればよい．

アドレス登録に対する不正　提案方法では，ユーザは，KP body を計算機にインストールする際にのみネットワークアドレスの登録を変更できる．しかし，鍵の不正共有を試みる者が，KP body のインストール時以外にもアドレスの変更ができるように KP body を改変する可能性がある．例えば，アドレス *Adr* の変更要求を，KP fragment 実行の度に送信するように KP body を改変することが考えられる（図 15参照）．改変された KP body は，どのアドレス上の計算機で実行しても瞬時にアドレスの登録が変更される．この KP body を複数の計算機上で同時に実行した場合，KP center へのアクセスが全く同時に発生しない限りは，それぞれの計算機上で KP fragment の実行結果を受け取ることができ，複数

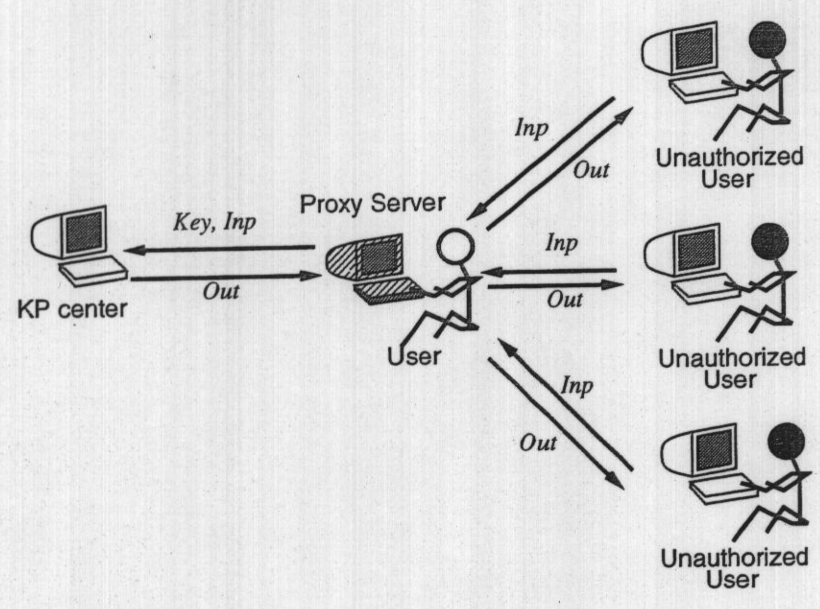


図 14 プロキシサーバによる不正

の人間が 1 個の鍵を共有することが可能となる．

この不正利用は，KP center へのアドレス変更要求の頻度を常にチェックすることで防ぐことができる．もし，アドレス変更要求の頻度が極端に高ければ，該当する鍵の使用を KP center が停止するなどの処置をとればよい．

2.5 Pay per use への応用

2.5.1 Pay per use の概念

Pay per use とは，プログラムを無料又はごく安価でユーザに配布する代わりに，使用頻度に応じてユーザに課金する方式である [33]．店頭でプログラム本体を販売する従来の方式と比べて，ユーザ，開発者の双方に利点がある．
[ユーザにとっての利点] ユーザは，使いもしないプログラムに大金を払うというリスクから解放される．ユーザは，プログラムを試用して気に入らなければいつでも使用をやめることができ，ごくわずかな金額だけを払えばよいことになる．
[開発者にとっての利点] プログラムが無料又はごく安価でユーザに配布されるた

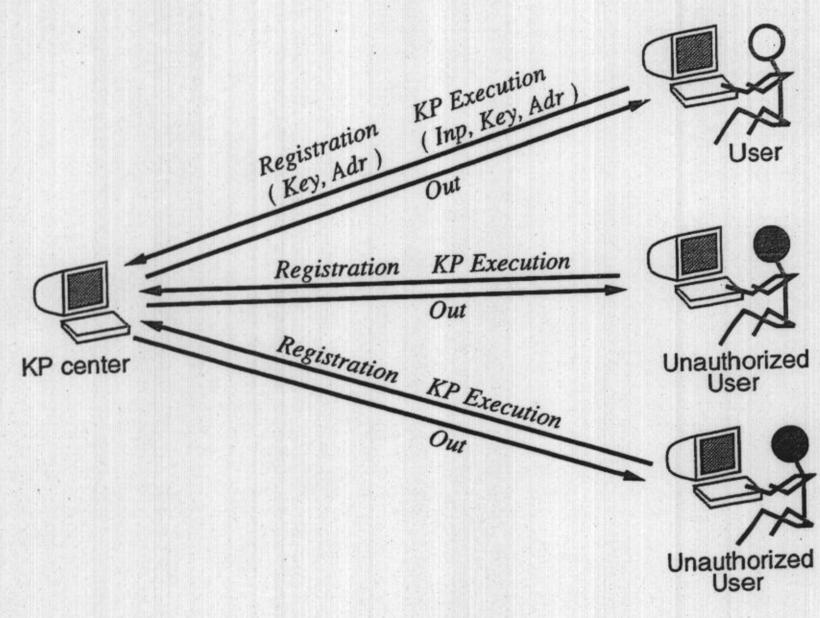


図 15 アドレス登録に対する不正

め，開発者は，より多くのユーザにプログラムを試用してもらうことができる．インターネット上でプログラムを流通させることで，流通コストも小さくできる可能性がある．また，プログラムの使用頻度に応じて課金されるため，多くのユーザに継続して使ってもらえるような良いプログラムを開発した場合には，莫大な利益を上げられることになる．

2.5.2 安全な KP による pay per use の実現

Pay per use システムはユーザに課金を行うため，極めて高い安全性が要求される．例えば，プログラム使用履歴の解析や改変を防止したり，ユーザがプログラムの使用頻度を偽るような不正を防ぐことが必須である．

本論文で提案した方法では，プログラムの使用頻度は，KP center へのアクセス回数として KP center 上でユーザ毎に記録できる．このアクセス記録はユーザの計算機上にはないため，ユーザが記録を解析したり改変したりすることはできない．また，プログラム（KP body）の実行には KP center へのアクセスが必須

となるため，ユーザがアクセス回数を偽ることも防止される．

Pay per use を実現する一つの方法として「超流通」システムが提唱されている [33][40]．しかし，安全な超流通システムの構成のためには，ソフトウェアの不正利用を防止し課金情報を管理するための専用ハードウェアを予めユーザに配布する必要があり，現状では必ずしも実用段階にない．一方，提案方法を用いた場合には，特殊なハードウェアを用いることなく pay per use の仕組みを安全に実現することができる．

2.6 まとめと今後の課題

本章では，安全な KP を提案し，従来の KP では防ぐことのできなかった不正利用をも防止できることを示した．また，提案方法の pay per use システムへの応用についても述べた．

安全な KP は，KP 全体を KP center に置くことによっても実現可能である．例えば，KP center に WWW サーバと CGI プログラム [10] を設置し，ユーザが WWW ブラウザを通してその CGI プログラムにアクセスすることによっても，安全な KP が実現できる（図 16参照）．ただし，KP center やネットワークの負荷が極めて大きくなり，ユーザが KP center から十分なサービスを受けられなくなる恐れがある．一方，提案した方法では，実行頻度の小さいごく小さなプログラムの断片（KP fragment）だけを KP center に置く．KP center やネットワークの負荷は極めて小さく，より多数のユーザへのサービスが可能である．

残された課題としては，障害により KP center が停止した場合にも継続してサービスが提供できるように，予備の KP center を設ける方法について検討する必要がある．また，より多くのユーザからのアクセスに対応するために，複数の KP center を用いて負荷を分散する方法について検討することも，重要な課題の一つである．

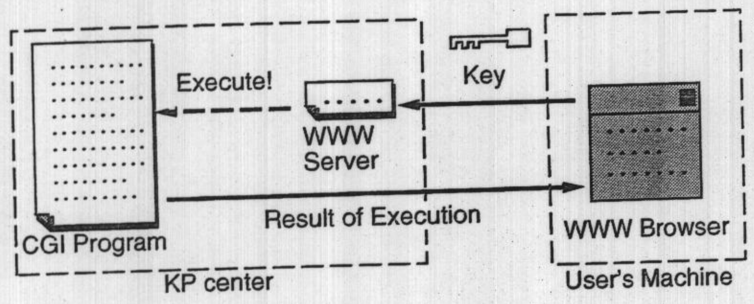


図 16 CGI プログラムによる安全な KP

3. プログラムに対する電子透かし法

3.1 あらまし

本章では，プログラムのコピー，プログラムの改変，および，部品の盗用の抑止を目的として，プログラムに電子透かし (watermark) を挿入する一手法を提案する．電子透かしとは，画像データやテキスト文書などの著作物をユーザに販売したり配布する前に，それらの著作物の中にあらかじめ埋め込んでおく情報のことである [4]．電子透かしとして埋め込まれる情報には，データの開発者，所有者，配布者の署名や，ユーザ ID などがあり，万一，データが不正に再配布された場合に，データの所有権を主張したり，再配布した者を特定する場合に役立つ．現状では，画像，音声，テキスト文書などに電子透かしを挿入する方法が盛んに研究されているが [5][25]，プログラムに電子透かしを入れることも，プログラムの不正な再配布を抑止する上で大変有効であると考えられる [31]．

本章で提案する方法は，C，C++，JAVA などのコンパイラによって生成された機械語プログラムに対して，任意の文字列を電子透かしとして挿入できる．

本方法は，以下の特長を持つ．

- ユーザや第三者によって透かしが改変されにくい．
- 透かしの挿入方法や挿入ツールを一般に公開した場合にも，透かしの安全性が保たれる．
- ツールなどにより，プログラム中から透かしを自動的に取り出せる．
- プログラムの一部分だけが切りとられて盗用された場合にでも，その部分に透かしが残っていれば，透かしを取り出すことができる．
- 透かしの挿入によって，プログラムの時間的な実行効率は変わらない．

以降，**3.2** では，プログラムの電子透かしの必要性について述べる．**3.3** では，プログラムの電子透かしに要求される性質を整理し，**3.4** では，80x86，Pentium 等の機械語プログラムを対象とする，より一般的な電子透かし法を提案する．**3.5**

では，JAVAプログラムに特化した方法を提案する．**3.6**では，関連する研究について述べる．**3.7**は，まとめと今後の課題である．

3.2 電子透かしが有用となる場合

電子透かしは，ソフトウェアの違法コピーの抑止，改変プログラム及び盗用プログラムの発見，改変及び盗用の事実の証明，のいずれにも有用であると考えられる．

3.2.1 違法コピーの抑止

プログラムのユーザは，自分が購入したプログラムをコピーして他人に渡したり，コピーしたプログラムをインターネット上などで再配布，または，販売するといった海賊行為を行う場合がある．従来は，プログラムの開発者が，不正に再配布されたプログラムを発見したとしても，誰が再配布を行ったかを知ることは必ずしも容易ではなかった．プログラムの違法コピーがなくならない理由の一つは，コピーしたプログラムを他人に渡した（再配布した）としても，そのことをプログラムの開発者に知られにくいことである．

しかし，プログラムをユーザに配布する前に，あらかじめ各ユーザの識別子を電子透かしとして挿入しておくことで，違法コピーの抑止に役立つと考えられる．不正に再配布されたコピープログラムを発見した場合に，そのプログラム中から電子透かしを取り出すことで，再配布を行った者を特定することが可能となる．電子透かしの存在は，不正が発覚する恐れがあるという無言の圧力となるため，違法コピーの抑止効果が期待される．

3.2.2 改変・盗用プログラムの発見

現状では，プログラムの一部だけが盗用された場合には，元のプログラムの仕様と盗用プログラムの全体の仕様は必ずしも類似しないため，その発見は困難である．特に，盗用する部品を改変してからプログラムに組み込まれた場合には，その発見は著しく困難となる．

近年，JAVA 言語で開発されたアプリケーションやアプレットが増えており，プログラムの一部だけが盗用されるという問題はますます深刻になると予想される．JAVA アプリケーションや JAVA アプレットは，複数のプログラム部品 (クラスファイル) から構成されており，インターネットを通してユーザが JAVA アプレット中から個々のクラスファイルを抽出することは容易である．JAVA アプレットのユーザは，Mocha，DeJaVu，SourceAgain 等の逆コンパイラを用いることにより，クラスファイルからソースプログラムを導出することも条件によっては可能であり [20]，クラスファイルの盗用を未然に防ぐことは容易ではない．

改変プログラム，および，盗用プログラムを発見するためには，インターネット上を移動してプログラム中の透かしを検査する「透かし検査ロボット」を開発することが重要である．透かし検査ロボットは，インターネット上を移動して JAVA 等のプログラムを探す．そして，プログラム中の電子透かしを自動的に検査し，盗用プログラムを発見したならば，電子メール等により自動的に透かし挿入者に報告する．予めプログラム中に透かしを挿入しておき，このようなロボットを用いることで，万一，クラスファイルが盗用され，そのクラスファイルを組み込んだ JAVA アプレットが再配布された場合にも，透かし検査ロボットによって発見できる可能性がある．

3.2.3 プログラム改変・盗用の事実の証明

盗用や改変の疑いがあるプログラムを発見したとしても，相手が自分のプログラムを盗用，または，改変したという事実を立証することは必ずしも容易ではない．プログラムを盗用，または，改変した者が，「これは新規に開発したプログラムである」と主張する可能性がある．そのような場合，盗用プログラムや改変プログラムが無断で再配布されたとしても，損害賠償を請求することは必ずしも容易ではない．

しかし，プログラムの著作権者の署名を，簡単な改変程度では容易に消去できないような形でプログラムの各部に挿入しておくことで，盗用や改変の事実の立証に役立つ．盗用プログラムや改変プログラム中に電子透かしが残っていれば，その透かしを取り出すことで，プログラムの正当な著作権者を明らかにすること

ができる．

3.3 プログラムの電子透かしに要求される性質

プログラムの電子透かしに要求される性質としては，以下の要件が考えられる．

(性質 1) 透かしが改変されにくい．

プログラム中の電子透かしは，悪意のあるユーザや第三者によって容易に改変されにくいことが望ましい．そのためには，透かしが埋め込まれている場所をユーザや第三者に容易に特定されないことが重要である．

(性質 2) 透かしの挿入方法が公開できる．

透かしの安全性を保つためには，透かし挿入方法から透かしを消す方法が容易に導き出せないことが要求される．透かしを消す方法が容易に導き出せる場合には，電子透かし挿入法を論文や特許公告等で一般に公開した時点で，透かしの安全性が保たれないことになる．例えば,「プログラム中の変数名や関数名に透かしの文字列を挿入する」というような透かしの挿入方法は，その方法が一般に知られた時点で，変数名をランダムに置き換える，という方法により透かしが消去できることも容易に知られてしまう恐れがある．

(性質 3) 透かし挿入ツールが公開できる．

透かし挿入ツールは，透かしを消すことに悪用されないことが要求される．一般に，すでに透かしの挿入されている（画像などの）データに対して，透かし挿入ツールを繰り返し適用することで，透かしが上書きされてしまう場合がある．

(性質 4) プログラムの実行効率を下げない．

透かしの挿入によって，プログラムの時間的な実行効率が変わらないことが望ましい．

(性質 5) プログラム変換ツールに対して耐性を持つ.

リエンジニアリングやプログラムの難読化などを目的とするプログラム変換ツール [11][12][34][43] によって，透かしが消えないことが望ましい．これらのツールは，電子透かしの消去に悪用される恐れがある.

(性質 6) 結託攻撃に対して耐性を持つ.

ユーザの識別子をプログラムに挿入する場合には，ユーザの結託攻撃 [42] に強いことが重要となる.

(性質 7) プログラム中から透かしを自動的に取り出せる.

プログラムコードを人間が読んだり解析することなく，ツール等によって透かしを自動的に取り出せることが望ましい．透かしが自動的に取り出せることは，透かしを取り出す者の負荷を小さくするだけでなく，透かし検査ロボットの実現にも役立つ．前節で述べたように，透かし検査ロボットが実現できれば，インターネット上の盗用プログラムや改変プログラムの発見に役立つ.

(性質 8) 切りとりに対して耐性を持つ.

プログラムの一部分だけが切りとられて盗用された場合にでも，その部分に透かしが残ってさえいれば，透かしを取り出せることが望ましい．一般に，画像データなどの電子透かしでは，データの一部分だけが切りとられて盗用された場合には，たとえ盗用された部分に透かしが残っていたとしても，透かしの取り出しが困難になる場合がある.「透かしが残ってさえいれば取り出せる」ということを実現すべきである.

3.4 電子透かしを挿入する方法

提案する透かし挿入方法の概略を図 17に示す．まず，コンパイル前のソースプログラムに対して，実際には実行されないステートメントの集合 (透かし挿入部) を追加する．次に，透かし挿入部を追加したプログラムをコンパイルする．そし

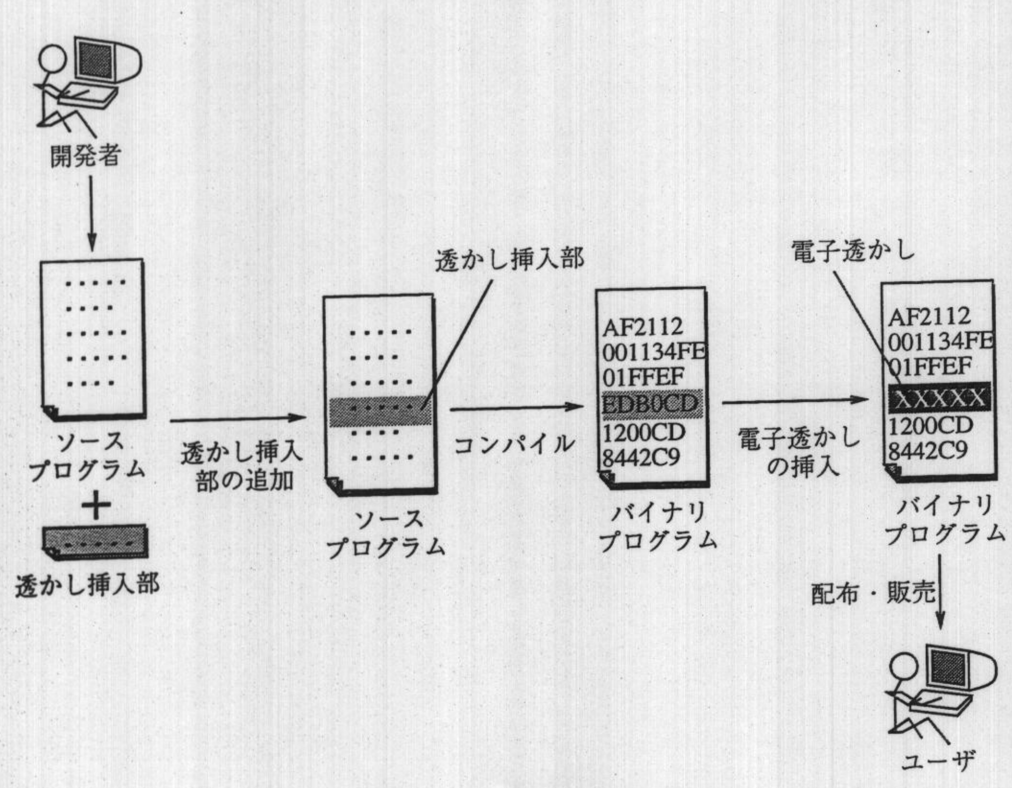


図 17 提案する電子透かし法の概略

て，得られた実行プログラム中の透かし挿入部に該当する部分の機械語コードに変更を加えることにより，透かしを挿入する．

本方法では，透かしの挿入時に新たに透かし挿入部が追加されるので，透かし挿入ツールを繰り返し適用した場合にも，以前から入っている透かしには影響を与えないため，性質 3 が満たされる．また，実行されない部分のプログラムにのみ透かしが挿入されるため，性質 4 が満たされる．

提案する電子透かし法は，次の四つの手順から構成される．

1. 透かし挿入部の追加

2. 目印の追加

3. 透かしの埋め込み

4. 透かしの取り出し

以降の節では，上記の 1〜4 の具体的な方法を述べる．

3.4.1 透かし挿入部の追加

プログラムに対する電子透かし法に要求される性質 1, 2 を満たすためには，透かし挿入部の位置がユーザに特定されにくいことが重要である．コンパイル後の機械語プログラム中の透かし挿入部の位置がユーザに特定された場合には，透かしが消されたり改変されたりする恐れがある．

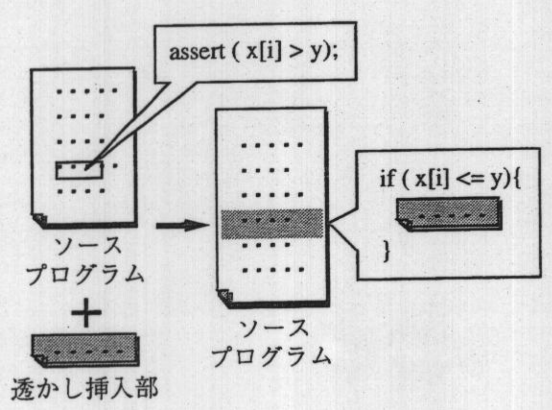


図 18 “assert” から “if” への変換

このことを実現する方法の一つとして，assert 法 [15] を用いる．assert 法は，プログラム中の assert 文を，必ず偽になる if 文に置き換え，その if 文の中身に実際には実行されないステートメントを追加する方法である．以下では assert 法を紹介する．

C，C++言語におけるデバッグ手法として，プログラマは，次のような assert マクロをソースプログラムに埋め込む手法が広く用いられる [37]．

```
assert(expression);
```

プログラマは，実行時に expression が必ず真になると仮定してプログラムを書く．したがって，プログラムにバグがない限りは，expression は必ず真になる．

assert法では，プログラム中のassertマクロを必ず偽になるif文に置き換える．

```
if(!expression) { X }
```

ここで，“!expression”はexpressionの否定を表す．Xは新たに追加する任意のプログラムを表す．Xは実際には実行されないため，Xの部分の仕様を自由に変更しても，プログラム全体の仕様は影響を受けない．本稿では，コンパイル後のプログラム中のXに該当する部分を書き換えることで透かしを挿入する．assert法を適用した例を図18に示す．

assert法を用いることで，ユーザに透かし挿入部の位置を見破られにくくできると期待される．ユーザが透かし挿入部の位置を特定するためには，プログラム中の全てのif文について必ず偽になるかどうかを調べる必要があるが，一般に，あるif文が全ての入力に対して常に真になるかどうかを調べることは容易ではない．コンパイル後の機械語プログラム中の条件分岐を全て調べることは，さらに困難となる．

3.4.2 目印の追加

透かしを挿入する際には，コンパイル後のプログラム中のどの部分が透かし挿入部に該当するかを知る必要がある(図19).

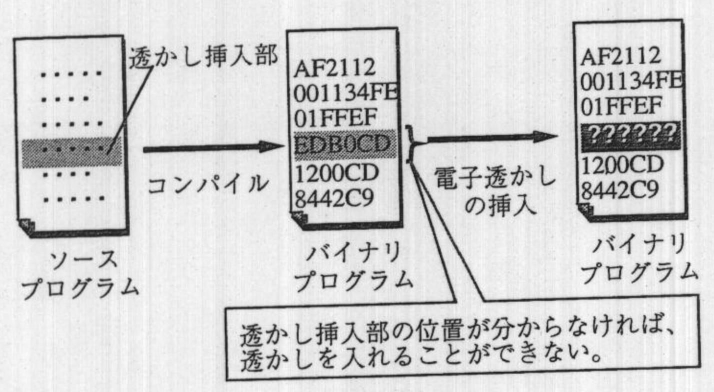


図 19 目印の必要性

ここでは一つの方法として，ソースプログラム中の透かし挿入部の開始位置と終端位置にそれぞれ目印をつけておくことにより，コンパイル後のプログラム中のどの部分が透かし挿入部に該当するかが容易に分かるようにしておく．BSD/OS 上の gcc 2.7.2 における目印の例を図 20に示す．コンパイル後の機械語プログラムにおいて目印となる機械語命令の組を見つけることで，透かし挿入部に該当する部分の開始位置と終了位置が自動的に特定できる．これらの目印となる機械語コードは，ユーザに電子透かしの位置を特定されて透かしが消されないように，透かし挿入時に適当な命令を上書きして消しておく．

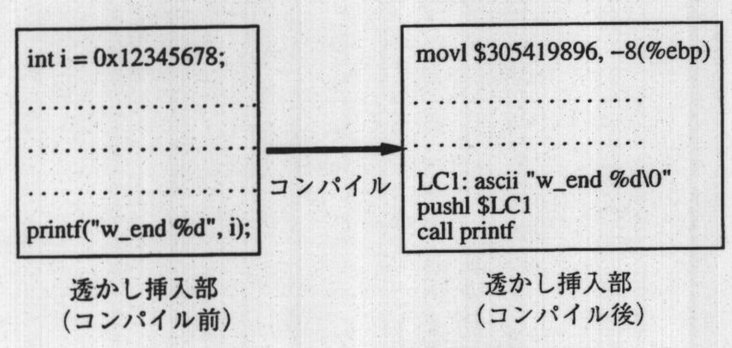


図 20 目印の例

透かし挿入部の内部に記述する内容には特に制限はないが，後で透かしを挿入するのに十分な量のステートメント (プログラム) を記述しておく必要がある．図 20の例の場合は，i に関する任意の計算を行うプログラムを書いておくことが考えられる．透かし挿入部の例は，**3.5.4** において図 27に示す．

3.4.3 透かしの埋め込み

透かしを埋め込む際に考慮すべきことは，透かしを挿入した後のプログラムが不自然にならないように配慮することと，性質 5 (プログラム変換ツールに耐性を持つ) を満たすことである (図 21).

透かしを挿入する最も単純な方法は，埋め込みたい文字列を，そのまま透かし挿入部の機械語コードに上書きすることである．しかし，このような単純な方法

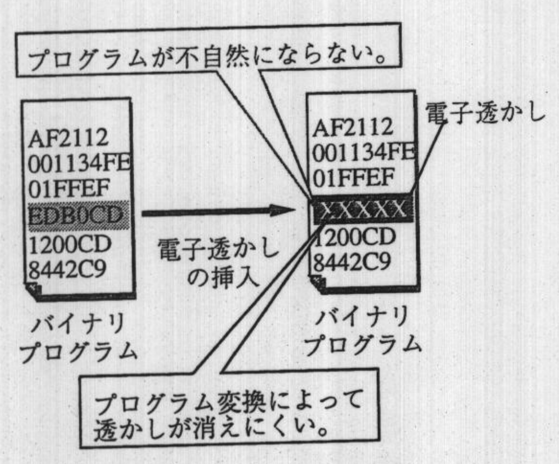


図 21 透かしの埋め込み

を用いると，機械語の命令セットに存在しない数値を書き込んでしまうなど，情報を埋め込んだ部分が機械語プログラムとして不自然になるため，情報を埋め込んだ位置がユーザに容易に特定される恐れがある．

JAVA 言語の場合は，さらに注意が必要である．JAVA アプレットは，実行される直前に，バイトコード検証器 (ベリファイア) によって文法や型のチェックが厳密に行われる [22]．そのため，透かし挿入部のプログラムが文法的に正しくないと，プログラムの実行が拒否される．.

本論文では，プログラムの構文上の正しさを保持したままで透かしを書き込む方法を提案する．以下では，まず，検証器によるチェックが行われない 80x86/Pentium CPU の機械語プログラムを対象とする方法を述べる．検証器のチェックを考慮した JAVA プログラムに対する方法は，次章で述べる．

提案する方法では，情報の埋め込み場所を，数値オペランドの部分に限定する．機械語プログラムの例を図 22に示す．図中，網掛けの部分が数値オペランドである．このプログラムでは，網掛けの部分に数値を上書きすることで，合計 6 バイトの数値 (電子透かし) を埋め込むことができる．

数値オペランド部分のみに情報を埋め込むことで，透かしを埋め込んだ部分のプログラムが構文上の正しさを保つことが保証でき，プログラムが不自然になることをある程度抑えることができる．

なお，透かしの入ったプログラムに対して，コードの最適化などのプログラム変換が行われた場合には，一部の数値オペランドの順序が入れ替わったり，消えたり，新たな数値オペランドが追加される可能性がある．したがって，性質5を満足するためには，透かしの文字列の一部の順序が入れ替わったり，削除されたり，雑音が付加された場合にも，なるべく元の情報が復元できるように，例えば誤り訂正符号化方式を併用することが望ましい．符号化の具体的な方法について検討することは，今後の重要な研究課題の一つである．

また，性質6（結託攻撃に耐性を持つ）を満たすためには，難読化ツールなどのプログラム変換ツールを利用して，多くのユーザに配布するそれぞれのプログラムに対して，透かしの入っていない部分をそれぞれ異なる表現に変換しておくことが有効である [15].

アドレス	機械語コード	ニーモニック
0000	B0 1A	MOV AL, 1Ah
0002	B9 23 01	MOV CX, 0123h
0005	BF 10 01	MOV DI, 0110h
0008	CD 20	INT 20h

図 22 80x86/Pentium のバイナリプログラム例

3.4.4 透かしの取り出し

透かしの取り出しにおいては，性質7（プログラム中から透かしを自動的に取り出せる），および，性質8（切りとりに対して耐性を持つ）を満たすことが重要である．盗用の疑いのある機械語プログラム中から透かしを取り出す場合には，透かしの存在の有無，および，プログラム中の透かしの位置は必ずしも明らかではないことに注意する必要がある (図 23).

本方法では，透かしがプログラム全体に入っているとみなして，プログラム中の全ての数値オペランドを取り出して並べる．すると，実際に透かしが入っている部分についてのみ，透かしの文字列が現れることになる (図 24).

ただし，誰でも自由に透かしが取り出せることにすると，プログラム中の透か

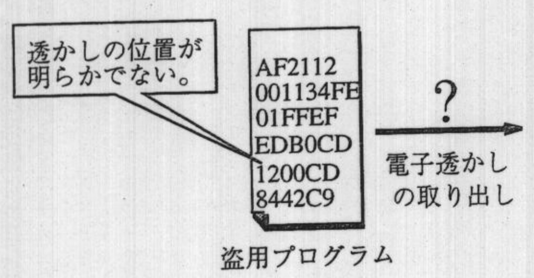


図 23　電子透かし取り出しの問題

し挿入部の位置がユーザに知られてしまい，透かしが消されたり改変される恐れがある．したがって，透かしとして埋め込む情報は，暗号化してから埋め込むことが望ましい．

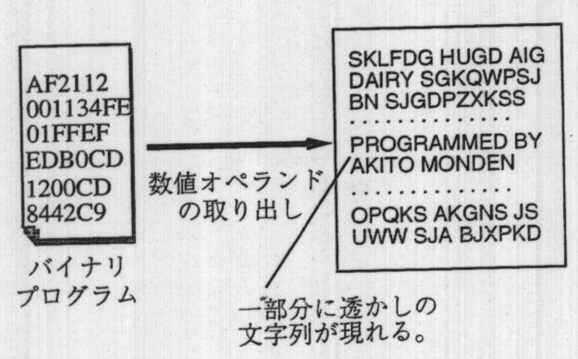


図 24　数値オペランドの取り出し

3.5　JAVA プログラムに対する電子透かし法

前節で述べた電子透かし挿入法は，JAVA プログラムにも適用が可能であるが，JAVA 固有の性質のため，いくつかの実用上の注意が必要である．以降では，前節で述べた方法をさらに拡張して，JAVA プログラムを対象とする具体的な電子透かし挿入法を提案する．

3.5.1　透かし挿入部の追加

3.4.2 節では，透かし挿入部を記述する際には目印をつけておく必要があると述べたが，JAVA 言語の場合は目印を入れておく必要がない．JAVA 言語では，ソースコード上の各メソッドの位置と，バイトコード (JAVA の中間コード) 上の各メソッドの対応付けが容易であるため [23]，個々のメソッドの開始位置と終了位置が目印となる．そこで，JAVA 言語においては，実際には実行されないメソッドを，透かし挿入部としてソースプログラムに追加し，メソッド内部のコードを書き換えることによって透かしの挿入を行う．

ただし，単に実行されないメソッドを追加しただけでは，そのメソッドがどのクラスからも呼び出されないことから透かしの位置をユーザに特定される恐れがある．そこで，3.4.1 で述べた assert 法などにより必ず偽になる if 文を作成し，その if 文中で追加したメソッドを呼ぶようにしておく．

3.5.2　透かしの埋め込み

JAVA バイトコードの場合は，自由に書き換えてもよいと思われる数値オペランドは，*iinc*，*bipush* などのごく一部の命令のオペランドに限られる．例えば，*getfield* や *putfield* などの命令の数値オペランドを自由に書き換えると，プログラム（アプレット）の実行時にバイトコード検証器のチェックをパスしなくなるため，プログラムが実行できなくなる．

しかし，書き換え可能な数値オペランドの部分だけに情報を埋め込むことにすると，極めて小さい情報量しか埋め込むことができないという問題がある．

そこで，検証器の検査をパスする範囲で，数値オペランド以外の部分のバイトコードも書き換えることにする．例えば，スタック内の int 型の要素の足し算 *iadd* は，他の int 型の演算命令 (*isub, imul, idiv, irem, iand, ior, ixor*) に置き換えても検証器の検査をパスする．つまり，*iadd, isub, ..., ixor* の 8 命令は互いに可換である．この性質を利用して，バイトコード中に *iadd* が現れた場合は，可換な 8 命令のいずれかに置き換えることで，3 ビットの情報を埋め込むことにする．例えば，*add* を 000_2，*isub* を 001_2，*imul* を 010_2，…，*ixor* を 111_2 にそれぞれ割り当てることで，000_2～111_2 の情報が表現できる．

表 1 命令コードに対する情報の割り当て例

命令コード	ニーモニック	情報
0E	dconst_0	0
0F	dconst_1	1
C6 xx xx	ifnull	0
C7 xx xx	ifnonnull	1
9B xx xx	iflt	00
9C xx xx	ifge	01
9D xx xx	ifgt	10
9E xx xx	ifle	11

演算命令以外にも，if 文や const 文などには可換な命令が存在する．それらの命令は，可換な命令の個数によってそれぞれ 1~3 ビットの情報の割り当てが可能である．命令コードに対する情報の割り当ての例を表 1に示す．

バイトコードに対して文字列を埋め込む場合には，文字列をビット列に変換し，ビット列の先頭から順に情報の埋め込みを行う．“ABCDE” の 5 文字 (5bit/1 文字) をバイトコードに埋め込んだ例を図 25に示す．この例では，各アルファベットを，‘A’$= 00000_2$，‘B’$= 00001_2$，‘C’$= 00010_2$，‘D’$= 00011_2$，‘E’$= 00100_2$ に割り当て，ビット列 “0000000001000100001100100” を埋め込んでいる．

3.5.3 プログラム変換に対する対策

前節で述べた方法は，数値オペランド部分のみに情報を埋め込む場合と比べると，コードの最適化や難読化などのプログラム変換に対する透かしの強度が著しく弱い．プログラム変換によって透かしが挿入されている部分のビット列が 1 ビットでもずれた場合には，正しい透かしの文字列が取り出せないことになる．例えば，バイトコード中に現れる “*iinc x y*” という命令を “*wide iinc 0 x 0 y*” という命令に無条件に置き換えてもプログラムの仕様に影響を与えないが，このような置き換えが行われると透かしが 8 ビットずれることになる．

アドレス	命令コード	ニーモニック	電子透かし
1000	02	iconst_m1	00 A
1001	60	iadd	000
1002	60	iadd	B 000
1003	68	imul	010 C
1004	3E	istore_3	
1005	84 01 21	iinc 01 21	00100001 D
1008	1C	iload_2	
1009	10 90	bipush 90	10010000 E

図 25 透かしを埋め込んだバイトコードの例

この問題を解決する一つの方法として，透かしのビット列がずれた場合に，それを補正するための“安全弁”となる命令 (例えば，*bipush*) を決めておくことが考えられる．透かしの埋め込み時に，安全弁となる命令に出会うたびに，現在埋め込み中の文字の 1 ビット目に戻って埋め込みをやり直すことにする．前節の透かし挿入例 (図 25) に安全弁による補性を適用した例を図 26に示す．図 26では，文字‘D’の挿入中に *bipush* が現れたため，‘D’の 1 ビット目から埋め込みをやり直している．この方法によって，透かしが挿入されている部分のビット列に雑音が付加された場合でも，*bipush* が出てくる度に補正されるため，雑音の付加による被害を軽減できる．

なお，このような安全弁を設けた場合には，必ずしもメソッドの先頭から透かしを埋め込む必要がない．メソッド中の任意の安全弁の位置から透かしの埋め込みを開始した場合にも，**3.4.4** で述べた方法を用いて透かしの取り出しを自動的に行うことが可能である．

3.5.4 透かしの挿入例

JAVA プログラムに対して透かしの挿入と取り出しを行うツールを作成し，文字列の挿入を行った．透かし挿入部の例として図 27に示すメソッドをソースプログラムに追加しコンパイルした．

得られたクラスファイル中の透かし挿入部に該当する部分は，178 バイトであった．このプログラム対して，付録に示すバイトコード変換規則を用いて透かしの

アドレス	命令コード	ニーモニック	電子透かし
1000	02	iconst_m1	00 A
1001	60	iadd	000
1002	60	iadd	B 000
1003	68	imul	010 C
1004	3E	istore_3	
1005	84 01 21	iinc 01 21	00100001
1008	1C	iload_2	
1009	10 19	bipush 19	D 00011001 E

図 26 安全弁による補正を考慮した透かしの例

挿入を試みたところ，最大 161 ビットの電子透かしの埋め込みが可能なことが分かった．この例では，平均すると 1 バイトあたり約 0.9 ビットの情報が表現できることになる．

```
private void x(int k){
    int i, j;
    for(i = 0; i < 10; i++)
        for(j = 0; j < 10; j++) k+=i*10+j;
    System.out.println("k = " + k);
    for(i = 0; j < 20; i++)
        for(j = j < 30; j++) k+=i*3-j;
    System.out.println("k = " + k);
    for(i = 0; i < 25; i++)
        for(j = 0; j < 20; j++) k+=i*4-j*3
    System.out.println("k = " + k);
}
```

図 27 追加するメソッドの例

3.6 関連する研究

プログラムに対する電子透かし法は，少数ではあるが，いくつか提案されている．

文献[13]において，C言語などのソースプログラム中の変数名を書き換えたりダミーの変数宣言を追加することで，任意のビット列を電子透かしとして埋め込む方法が紹介されている．しかし，この方法では，プログラム中の変数名をランダムに置き換えることで容易に透かしが消されてしまうという問題がある．また，実行プログラムには必ずしも適用できない．

文献[18]では，JAVAプログラム中の配列変数に任意のバイト列を埋め込む方法を提案している．この方法は，プログラムが改変された場合にも透かしが消えにくいという特長がある．しかし，透かしを取り出す際には，プログラム中の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置き換える必要がある．そのため，盗用プログラムを自動的に発見することはできない．また，プログラムの一部分だけが盗用された場合には，透かしを取り出すことが困難となる．

文献[15]では，C言語などのソースプログラムに対するいくつかの電子透かし法を紹介しており，中でも，assert法が有力であると述べている．ただし，具体的な電子透かし挿入方法は提案していない．本論文で提案した電子透かし挿入方法は，このassert法を，実行プログラムに対して応用した方法である．

3.7 まとめと今後の課題

C，C++，JAVAなどのコンパイラによって生成された機械語プログラムを対象とする電子透かし法を提案した．本章の前半では，80x86，Pentium等の機械語プログラムを対象とする，より一般的な電子透かし法を提案し，後半では，JAVAプログラムに対して効率的に電子透かしを挿入する方法を提案した．また，透かしの挿入例を紹介した．

本方法は，プログラムの一部分だけが盗用された場合など，一見しただけでは盗用プログラムかどうかの判断が困難な場合にも，盗用プログラム中に透かしが残っていれば，透かしを取り出すことができる．また，透かしの取り出しが自動的に行えるため，透かし検査ロボットによって，インターネット上で再配布された盗用プログラムを自動的に発見することが可能である．

3.4.4で述べたように，透かしの挿入時には，透かしの内容を暗号化してから埋め込むことが望ましい．ただし，任意の暗号化アルゴリズムを自由に用いて透

かしを埋め込むことにすると，実際の裁判において,「元から透かしが入っていないプログラムに対して，さも透かしが入っていたかのように，暗号化アルゴリズムを後から恣意的に作成したのではないか」という主張を崩せない恐れがある．

この問題に対する一つの解決方法として，透かしの挿入を行う者は，自分が用いた暗号化アルゴリズムの種類や鍵の内容を公の機関にあらかじめ登録しておき，裁判になった場合にのみ，その機関が暗号アルゴリズムと鍵を公開するという方法が考えられる．

今後の課題としては，透かしの強度の評価，および，より耐性のある透かし挿入法について検討していく必要がある．

4. プログラムの難読化法

4.1 あらまし

ソフトウェアの保守，再利用などのためには，解析や理解の容易なプログラムを作成しておくことが重要である．ところが，プログラムを多数のユーザへ配布する場合には，逆に，内部の解析が困難なプログラムの作成が要求される場合がある．それは，**4.2.2** 節で述べるように，プログラム内で使用されている方式やアイデアなどが，そのプログラムの解析によって，ユーザや第三者に漏洩することを防止したい場合である [26][27][28][34]．

保守，再利用などの効率を損なわずに，しかも，プログラムの解析を困難にするためには，図28のような方式が有用であると考えられる．まず，解析が容易になるように開発者がプログラムを作成する．次に，作成したプログラムを残しておいて，そのコピーを，解析が困難になるように何らかの方法で変換する．そして，解析が困難な方のプログラムをユーザへ配布する．解析が容易な方のプログラムは，保守，再利用などのために開発者が保管する．このようなプログラムの等価変換を，**プログラムの難読化**と呼ぶ．

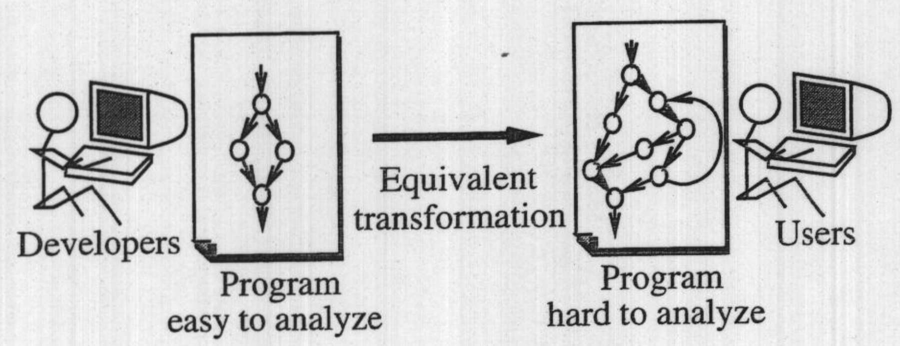


図 28 プログラムの難読化

難読化の対象は，機械語のプログラムとその他のソースレベルの言語で書かれたプログラムとの両方である．ユーザに配布されるプログラムは，機械語で書かれている場合が多いが，ソフトウェアの形態の多様化に伴って，ソースレベルで配

布される場合も増えているからである．例えば，UNIX のフリーウェア，JAVA スクリプト，Perl のスクリプトなどは，ソースレベルでネットワーク上で配布されることが多い．Smalltalk やその他のインタプリタ言語で書かれたプログラムも，ソースレベルで配布されることがある．また，条件によっては，リバース・エンジニアリング [6] の技術によって，機械語のプログラムからソースレベルのプログラムが復元されることもある．

本章では，ループを含むプログラムを自動的に難読化する方法を 2 通り提案する．提案する方法は，ループを含むプログラムであれば，ソースレベルでもアセンブリ言語でも適用できる方法である．そして，方法の有効性を評価するための実験の結果について報告する．関連する方法としては，文献 [34] に自動難読化ツールが紹介されている．このツールは，アセンブリ言語のプログラムを対象としており，複雑な命令を単純な命令の列に書き換えたり，命令を並べ換えたりする．提案する難読化法は，プログラム中のループの部分を複雑に変換し，ソースレベルのプログラムにも適用できる．

4.2 難読化が有用となる場合

プログラム内で使用されている方式やアイデアがプログラムの解析により漏洩することを防止したい場合としては，以下の場合が考えられる．

4.2.1 知的財産権を保護したい場合

プログラム内で使われる方式やアイデアが独創的で価値が高ければ，それを考案した者が，それを所有する権利を主張することがある．そのような権利については，利害の対立が発生することがあるので，何らかの保護が必要になる．特に，ソフトウェアが売買される場合には，必要である．ところが，現状では，ソフトウェアに関する権利を保護するための法的な体制は，十分に整備されていない [19][24]．また，方式やアイデアが盗用されたとしても，それを立証することは容易でない．そこで，そのような盗用を防止するためには，プログラム自身に工夫を施すことにより，方式やアイデアをユーザや第 3 者に対し隠蔽しておくことが重要であると考えられる．つまり，プログラムの難読化が有用であると考え

られる．

4.2.2 システムに対する不正を防止したい場合

プログラムの難読化は，プログラム開発者の知的財産権を保護するという目的以外にも，システムに対する不正を防止したい場合に役立つ．

プログラムが組み込まれているシステムに対して，悪意を持つユーザが不正操作を行おうとすることがある．もし，プログラム内で使用されている方式やアイデアがその者に容易に知られたとすれば，不正を許す危険性が増すことになる．逆に，プログラムを難読化すれば，不正を許す危険性を軽減できると言える．

例えば，認証システムに対して権限を持たない者が不正に認証を得ようとしている場合を考える．そのようなシステムは，文献 [35] で述べられているように，開発者の予想を越えるような方法による不正操作を見逃すことがある．もし，プログラムの解析により，認証方式の詳細がその者に知られたとすれば，不正の方法が発見される危険性が増すことがある．

また，文献 [34] では，Key Escrow System と言う暗号化システムの例が述べられている．そのようなシステムをソフトウェアによって実現する場合に，プログラムの解析により，符号化や復号のアルゴリズムが漏洩することがある．

4.3 難読化についての定義

難読化の目的は，プログラムの解析を困難にして，方式やアイデアが漏洩することを防止することである．そこで，まず，解析に対して定義を与え，次に，難読化に対して定義を与えることにする．

4.3.1 プログラムの解析

ここで言うプログラムの解析とは，ある者が，プログラム内で使用されている方式やアイデアを，他のプログラムで使用できる程度に知ろうとする行為である．従って，単に，プログラム中の各命令の意味を知ることではなく，より深い知識を得ようとする行為である．しかも，その深い知識とは，その者が知ろうとしている，あるいは，開発者が隠そうとしている方式やアイデアであるから，同一の

プログラムについても場合によって変わると考えられる．そこで，ここでは，そのような知識を命題として表すことにして，次の定義を与える．

プログラムの解析: プログラム Pと Pに関する命題 Q とが与えられた時，Q を論理的に証明することを，Q に関する Pの解析と言う．

4.3.2 難読化の定義

既に述べたように，プログラムの難読化とは，ユーザへ配布するプログラムの解析を困難にすることである．暗号化と似ているが, その方式や特徴は大きく異なる. 暗号化したプログラムは, 配布の過程での解析が困難であるが, そのままでは計算機が解釈できないので, 実行されるまでに必ず復号される. 従って, 多数のユーザに配布され不特定の計算機で実行される場合, 復号後に解析される危険性が残る. 一方, 難読化は等価変換であり, 難読化されたプログラムは, 復号のような逆変換を行なわなくても実行でき, 変換前と同一の仕様を満たす．暗号化されたプログラムより解析が容易であるかもしれないが, 多数のユーザに配布される場合に有効である.

難読化したプログラムも暗号化したプログラムも, 無制限に時間を費やせば解析できないとは限らない．そこで, 現実的に不可能に近い程度に解析を困難にすることが重要である. 但し, どの程度の時間が必要であれば現実的に不可能であるかは, 現時点で明らかでない．そこで, 本論文における難読化では, 解析に少しでも長い時間を要するように変換することを目的とする.

なお, プログラムを変換すると，仕様を変えなくても, 大きさが増したり，実行効率が低くなったりすることが考えられる．近年では，記憶装置の容量が飛躍的に増加しているので，大きさの増加は，ある程度許容されると思われる. しかし，時間的な実行効率の低下は，必ずしも許容されない．計算機の速度も上がっているが，それ以上に，高速な処理を必要とする高機能なアプリケーションの需要が高まっているからである．

以上の議論に基づいて，次の定義を与える．

プログラムの難読化: ある言語で書かれたプログラム Pと Pに関する命題 Q が与えられたとする．その時に，同一の言語で書かれたプログラム P'を，次

の 3 条件を同時に満たすように導くことを，Q に関して Pを難読化すると言う.

(仕様の保存) 任意の入力について，P'は Pと同一の出力を返す.

(実行効率の保存) 任意の入力について，P'を実行した時の演算，代入，比較などの回数は，Pと同じ，または，少ない.

(解析の困難さの増加) P'は，Q に関する解析に Pよりも時間がかかる.

変換による大きさの増加については, 場合によって許容範囲が変わると思われるので, 定義中で明確な基準を述べていない.

4.4 ループを含むプログラムの難読化

提案する難読化の方法は，ループを含むプログラムを対象とする．例えば，for 文，while 文，repeat 文などを含むプログラムである．図 29は，C 言語の例であり，整数配列中の最大要素を求めるプログラムである．提案する方法は，そのようなプログラムを，入出力についての任意の命題に関して難読化する．例えば，図 29のプログラムを,「$N \neq 0$ の時の出力は，配列の最大要素に等しい」と言う命題に関して難読化する.

```
int M (int N, int A[]) {
    int x = 0;
    int i;
    for(i = 0; /* iteration starts */ i < N; i++) {
        if (A[i] > x) x = A[i];
    }
    return x;
}
```

図 29 ループを含むプログラム

ループを含むプログラムについてのそのような命題の証明では，一般に，ルー

プ不変式を発見することが重要であることが知られている [2][21]．従って，そのようなプログラムの難読化には，ループ不変式の発見を困難にすればよいと考えられる．ループ不変式とは，各繰り返しの開始時点で常に成り立つ命題，あるいは，式である．図 29のループについては，例えば，

$$i = I \;\Rightarrow\; x = \max(0, A[0], A[1], \ldots, A[I-1])$$

がループ不変式である．i のように，ループが繰り返される度に増減して，ループからの脱出の条件を判定する時に参照される変数を，制御変数と呼ぶ．

4.5 提案する 2 通りの難読化の方法

ループを含むプログラムを前章の方針に基づいて難読化する方法，つまり，ループ不変式の発見を困難にする方法を，2 通り，提案する．制御構造だけを複雑に変えることによりループ自身の存在を発見しにくくする方法と，繰り返される処理の順序を複雑に変える方法とである．

難読化は等価変換であるので, その逆変換も考えられる. 但し, 存在する逆変換が順方向の変換と同程度に容易であるとは限らない. 提案する方法では, どちらも, ループを容易に複雑化するが, 変換後のプログラムから元のループを復元することは大変に困難である.

4.5.1 制御構造を複雑化する方法

本方法では，ループにおいて繰り返される処理について，その内容も順序も全く変えない．その上で，繰り返し文や分岐文により表現される制御の構造だけを複雑に変える．例えば，図 29のプログラムに対しては,「$A[i] > x$ ならば $x \leftarrow A[i]$」と言う処理について，その内容も順序も変えない．その上で，for 文や if 文を，goto 文や for 文や if 文を組合わせて複雑に置き変える．

そのような変換のために，本方法では，プログラムを表す有向グラフを使用する．つまり，図 30に示すようなフローチャートを使用する．この図は，図 29のプログラムを表しており，4 種類の節からなっている．つまり，プログラムの開始節と終了節 (塗りつぶした小円)，条件分岐を表す分岐節 (菱形)，分岐文や条件文

の他の実行文を表す節 (矩形)，分岐節に対する合流節 (小円) である．なお，分岐節以外の 1 個の節から複数の辺が出ることは許されないし，合流節以外の 1 個の節へ複数の辺が入ることも許されないことにする．

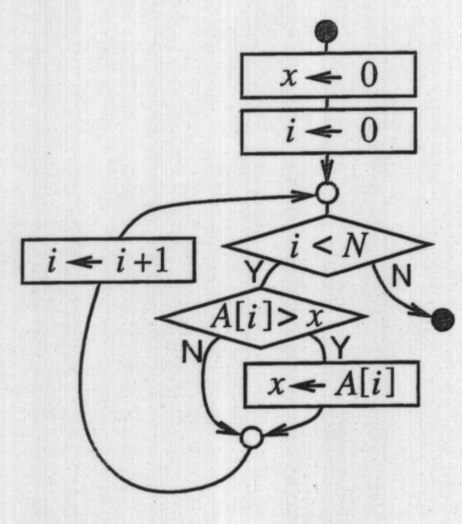


図 30 フローチャート

変換の手順 具体的な変換の手順は，次の通りである．

1. プログラムを有向グラフへ変換する．

2. 表 2に示す 4 個の変換規則をグラフの部分に適用することをランダムに繰り返し, グラフを次第に変形する．

3. 3 個以上の辺が入っている合流節が残っていないことを調べる．残っていれば，表 2の第 4 規則を適用して，無くなるように操作する．

4. 有向グラフをプログラムへ逆変換する．

各変換規則は, グラフ中のある形状の部分を別のある形状に置き換える．表 2では, 各変換規則を, 変換前の形状と変換後の形状との組で表している．変換前の形状に一致すれば, グラフのどの部分にでも適用できる．第 1 規則と第 2 規則は,

表 2 制御構造の変換規則

	1 ループ の移動	2 ループ の拡大	3 合流点 の併合	4 合流点 の分割
変換前	G_1 G_2	G_1		
変換後	G_1 G_2 G_1	G_1 G_1		

ループの部分を対象とする規則であり, 図に含まれる G_1や G_2は, 1 個以上の節からなる任意の部分グラフを表している.

例えば, 第 1 規則は, 1 個の合流節とその他の 2 個以上の節とからなる任意の閉路に適用できる. 合流点を除く部分が, G_1と G_2に分けて表されている. G_1や G_2には, どのような複雑なグラフが含まれてもよいし, 分岐節が含まれてもよい. また, 図では, この閉路へ外部から入る辺が 1 個だけ描かれ, 外部へ出る辺が描かれていないが, G_1や G_2には, 任意の個数の辺が入っていたり出ていたりしてもよい. ほとんどのループには, そのループから抜け出るための条件分岐があるが, そのような分岐のそれぞれは, G_1または G_2中の 1 個の分岐節とその節から外部へ出る辺に対応する.

第 1 規則を適用すると, G_1の部分が複製され, 合流節の位置が変わる. G_1から外部へ出る辺や外部から G_1へ入る辺があった場合の G_1の複製は次の通りである. まず, 外部から G_1へ入っていた辺については, 複製せずに, 複製後の 2 個の G_1のどちらか一方へ入るようにする. どちらを選ぶかはランダムである. そして, G_1から外部へ出ていた辺については, G_1と同様に複製する. 但し, 複製しただけでは, 外部へ出る辺の数が増えることになってしまうので, 1 個の合流点を追加して 2 辺を束ねるようにする.

4 個の変換規則の他にも変換規則が考えられるが, ほとんどの変換はこれら 4 個の組合わせとして表せる.

適用例　図 32は，図 29のプログラムに本方法を適用して得られた例である．フローチャートへ変換した後に，図 31のように第 1 規則を 2 回続けて適用し，プログラムへ逆変換した．破線で囲んだ部分は，表 2の第 1 規則の変換前のグラフにおける G_1と G_2に相当する. これは, 特別に解析が困難になった例ではない. 変換規則の適用をランダムに繰り返せば, 他にも様々な複雑なプログラムが得られる.

仕様の保存　本方法を構成する 4 段階の内，第 1，第 4 段階は，プログラムと有向グラフとの間で，意味を変えずに表現だけを変える変換である．第 2，3 段階は，表 2の単純な変換の繰り返しであるが，各変換は実行される節の順序も内容も全く変えない. 従って，本方法を適用しても，プログラム全体の仕様は保存さ

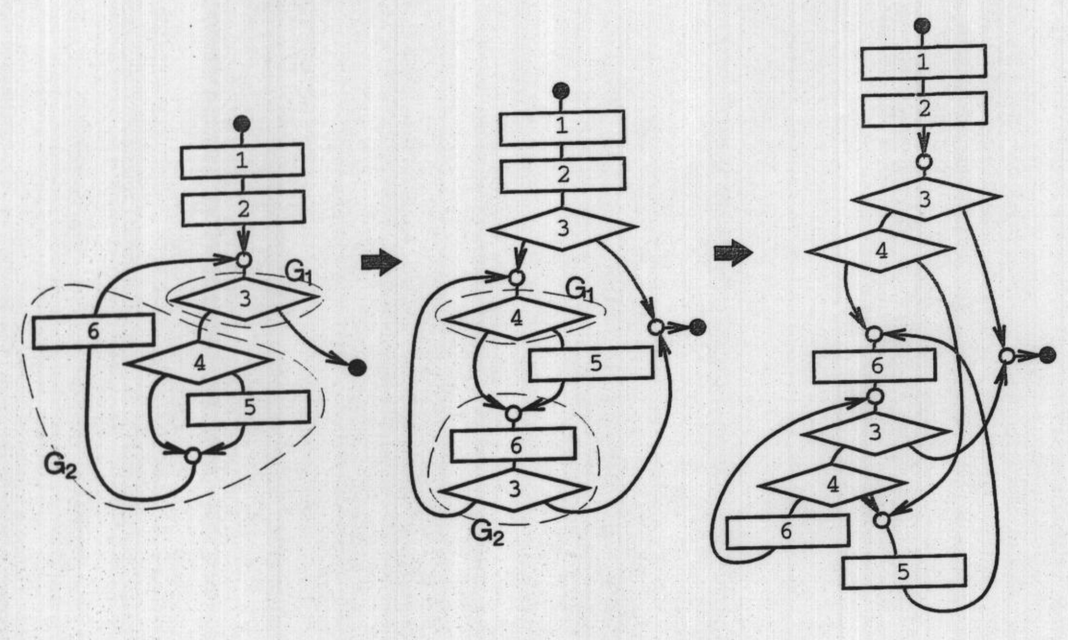


図 31 規則に基づく変換

れる.

実行効率の保存　本方法では，処理の内容も順序も変えない．従って，実行時の演算, 代入, 比較などの回数は, 全く変わらない.

大きさの増加　本方法では，第 2 段階における有向グラフの変換が原因で，変換前に比べて変換後のプログラムが大きくなることがある．それは，第 1，第 2 規則の適用により，グラフの部分が複製される場合である．それ以外の変換では，プログラムの大きさがほぼ変わらない.

プログラムの大きさが極端に増加することを避けたい場合には，第 2 段階において，第 1，第 2 規則の適用回数を制限する必要がある.

自動化　本方法を構成する 4 段階の内，第 2，第 3 段階は単純な変換であるので，自動化が容易である．また，第 1，第 4 段階ついても自動的に行う方法が既に提案されている [11]．従って，本方法の全体も自動化できる.

```
int M (int N, int A[]) {
    int x = 0;
    int i = 0;
    if (i < N) {
        if (A[i] > x) {
L1:             x = A[i];
        }
        i++;
        for(;;) {
            if (i >= N) break;
            if (A[i] > x) goto L1;
            i++;
        }
    }
    return x;
}
```

図 32 制御構造を複雑化したプログラム

4.5.2 処理の順序を複雑化する方法

本方法では，繰り返される処理の順序を入れ換えてもよいようなループを見つけて，その順序が複雑に入れ換わるように制御構造を変える．このような変換によって，繰り返しの度に 1 ずつ増加する，あるいは，減少するような発見されやすい制御変数を減らすことができる．例として，図 29のループを考える．このループにおいては，繰り返しの度に制御変数 i が 1 ずつ増加し,「$A[i] > x$ ならば $x \leftarrow A[i]$」と言う処理が繰り返されている．よく読むとわかるように，この処理の順序をどのように入れ換えたとしても，プログラムの出力は変わらない．そこで，本方法では，例えば，その処理が 0，5，1，4，3，2 の順に進むように制御構造を変える．

但し，一般には，そのように処理の順序を入れ換えようとすると，制御変数を増減させるための処理が増えことが多い．従って，実行効率を全く低下させずに，処理の順序を入れ換えることは，容易でない．そこで，本方法では，次の方式を採る．まず，難読化するプログラムが与えられる前に，効率を低下させないような変換をあらかじめ調査する．そして，そのような変換のパターンを可能な限り多く収集して，カタログとして整理しておく．そして，難読化するプログラムが与えられた時に，各ループに対して適用可能なパターンを調べる．

変換の手順　具体的な変換の手順は，次の通りである．

1. プログラムを有向グラフへ変換する．

2. プログラム中の個々のループに対して, 適用可能な変換パターンをカタログから選出しランダムに適用する．各変換パターンは, **4.5.1** の変換規則と同様に, プログラムのある形状の部分を別の形状に置き変え, そして, 変換前と変換後の形状の組で表される．ある変換パターンが適用可能であるとは, 形状が一致し, かつ, 各変換パターンに固有の適用条件を満たすことを言う．

3. 有向グラフをプログラムへ逆変換する．

これまでに筆者は 15 個の変換パターンを得ている．それらの変換パターンの例を表 3に示す．図に含まれる $G_1(i)$ や $G_2(j)$ は，1 個以上の節からなるプログラ

ムの部分である．変換前のプログラムにおける $G_1(i)$ や $G_2(j)$ は，繰り返される回数が固定されていて，かつ，順序を入れ換えてもよい処理を表す部分である．$G_1(i)$ や $G_2(j)$ が満たすべき性質は，自動化の所で述べる．i や jは，それらの部分で参照される制御変数を表している．また，$G_1(N)$ は，$G_1(i)$ 中で変数 i を参照している箇所を変数 Nの参照に置き換えたものを表している．$G_2(N)$ も同様である．

表中の 4 パターンの固有の適用条件は，以下の通りである．

- パターン 1
  $G_1(i)$ では Nは参照も更新もされない．また，パターンの部分の後で i および N の値が参照されない．

- パターン 2
  $G_1(i)$ では Nは参照も更新もされない．また，パターンの部分の後で i および N の値が参照されない．

- パターン 3
  とくになし．

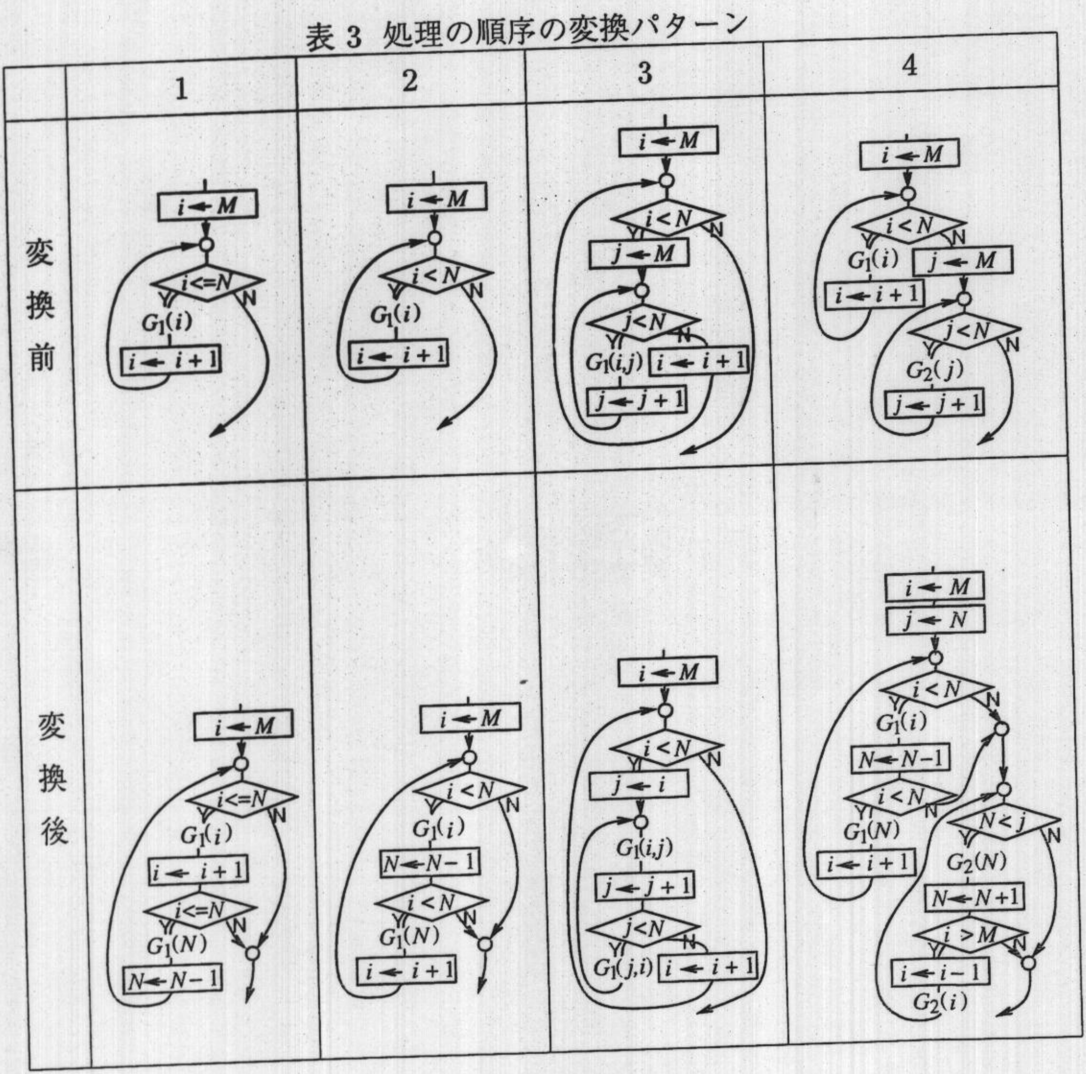
- パターン 4
  $G_1(i)$ および $G_2(j)$ ではでは Nは参照も更新もされない．

表に示していない残りの 11 個の変換パターンは，2 重ループを対象とする 7 パターンと，連続する 2 個以上の 1 重ループを対象とする 3 パターンと，1 個の 1 重ループを対象とする 1 パターンとである．

適用例　図 33は，図 29のプログラムに本方法を適用して得られた例である．変換前のループにおける繰り返しは，N回であるとあらかじめ容易に計算できる．繰り返される処理である「if($A[i] > x$) $x = A[i]$」は，順序を入れ換えてもよい．そこで，変数 Nがこのループの後に参照されないと仮定すると，パターン 2が適用可能になる．その適用の結果，図のプログラムが得られることになる．

また，本方法は，前述の制御構造の複雑化と併用することが可能である．但し，前方法を適用した後に本方法を適用することは，効果がない．なぜなら，前方法

表 3 処理の順序の変換パターン



```
int M (int N, int A[]) {
    int x = 0;
    int i;
    for(i = 0; i < N; i++) {
        if (A[i] > x) x = A[i];
        N--;
        if (i >= N) break;
        if (A[N] > x) x = A[N];
    }
    return x;
}
```

図 33 処理の順序を複雑化したプログラム

を適用すると，ループが複雑になるので，本方法の変換パターンを適用できる可能性がほとんどなくなるからである．本方法を適用した後に前方法を適用することは，効果があり，ループ不変式の発見を一層困難にすると思われる．図 34は，図 33のプログラムに，更に前方法を適用して得られた例である．

仕様の保存　カタログには，プログラムの仕様を必ず保存する変換パターンだけを入れておく．従って，本方法を適用しても，プログラムの仕様は保存される．表 3の各変換パターンについての証明は省略する．

実行効率の保存　仕様の保存についてと同様に，カタログには，実行効率を保存する変換パターンだけを入れておく．表 3の各変換パターンをよく見るとわかるように，どの変換パターンを適用しても，実行時の分岐条件の判定，代入，その他の処理の回数は，増加しない．変換パターンによっては，パターン 3のように，分岐条件の判定の回数が減少する．

```
int M (int N, int A[]) {
    int x = 0;
    int i = 0;
    if (i < N) {
        if(A[i] > x) {
L1:             x = A[i];
        }
        for(;;) {
            N--;
            if (i >= N) break;
            if (A[N] > x) x = A[N];
            i++;
            if (i >= N) break;
            if (A[i] > x) goto L1;

        }
    }
    return x;
}
```

図 34 処理の順序と制御構造を複雑化したプログラム

大きさの増加　表 3からわかるように，これまでに得られている変換パターンについては，どれを適用しても，変換前に比べて変換後のプログラムが大きくなる．但し，プログラムの大きさを文の個数で表すと，変換後の大きさは変換前の 2 倍以下の大きさである．

自動化　変換パターンの適用においては，変換前のプログラムにおける $G_1(i)$, $G_2(j)$，$G_1(i,j)$ に該当する部分 (以下，まとめて G_kと記す) が, 順序を入れ換えてもよい処理であるかどうかを，プログラムの静的解析によって判別する必要があるが，この判別は容易ではない．そこで, 現時点では, 下の 3 条件を満たすような判別の容易なループだけを考える．これによって, パターンを適用できるループの一部しか検出できないが, 自動的な検出が行える．

1. ループの外から G_kへ入る制御フローも，G_kからループの外へ出る制御フローも存在しない．

2. G_kでは，参照と更新の両方が行われる変数は存在しない．

3. G_kでは，配列やポインタへの値の代入は行われない．

以上の条件によって, 処理の順序を入れ換えてもよいことが保証される．この条件を緩めてパターンをより多く自動的に適用できるようにすることは, 将来の重要な課題である．

4.6　方法の評価実験

4.6.1　実験の方法

提案する方法により難読化したプログラムとしていないプログラムとに対して，被験者に解析を試みてもらう実験を行った．

用意したプログラムは，次の 3 種類，9 個である．

種類 0: 難読化していない 20 行程度のプログラム P_0，Q_0，R_0

種類 1: P_0，Q_0，R_0の制御構造を複雑化して得られた P_1，Q_1，R_1

種類 2: P_0，Q_0，R_0の処理の順序を複雑化した後に制御構造も複雑化して得られた P_2，Q_2，R_2

被験者は，9 名の大学院生である．各被験者に種類 0，1，2 のプログラムを 1 個ずつ割り当て，27 回の試行を実施した．

各回の試行においては，被験者に，プログラムリストと，そのプログラムに関する真の命題と，命題の証明を記入する用紙とを与えた．各命題は，「X を入力すると Y を出力する」のような形式であった．そして，その証明を試みてもらい，用紙への記入が終った時にそのことを自発的に宣言してもらった．証明の記入については，あらかじめ，具体的な方法を被験者に指導し，各被験者に 3 回以上の証明の演習を行ってもらっていた．記入された証明に間違いがあった場合には，間違いがあることだけを告げ，再度，証明を試みてもらった．そして，プログラムを与えてから，正しい証明が得られるまでに要した時間を測定した．

本実験では，各被験者が 3 回の試行を行ったので，適用した難読化法の他に，試行に対する被験者の慣れが，測定する時間に影響する危険性があった．そこで，被験者にプログラムを割り当てる際には，そのような慣れができるだけ評価に影響しないように，次の 4 点に注意した．第 1 に，各被験者について，P_0，P_1，P_2 の中から 2 個以上を割り当てないようにした．Q と Rについても同様にした．同一の仕様を持つプログラムを 2 度以上解析することによる慣れの影響を防ぐためである．第 2 に，種類 1 のプログラムの中から 2 個以上を割り当てないようにした．種類 2 についても同様にした．同一の方法を適用したプログラムを 2 度以上解析することによる慣れの影響を少なくするためである．第 3 に，各被験者について，1 回目の試行に種類 0 のプログラムを割り当てた．慣れの影響が，難読化していないプログラムの時間の平均を不当に短くすることを防ぐためである．第 4 に，約半数の被験者について，2 回目に種類 1，そして，3 回目に種類 2 のプログラムを割り当てるようにした．残りの被験者については，その逆にした．慣れの影響が，種類 1 と種類 2 のどちらかの時間の平均を不当に短くすることを防ぐためである．

4.6.2 実験の結果

表4に，各種類のプログラムについて，証明に要した時間と，証明に誤りがあった回数とを示す．プログラムの種類ごとに時間の平均を取った結果は，およそ1 : 4 : 18の比率で大きく異なっていた．証明に誤りがあった回数の平均も，0.11回，0.22回，3.44回と大きく異なっていた．これらの比率は，実験条件によって変動すると思われるが，ごく小規模なプログラムに対しても，難読化によって桁の変わる程に解析を困難にできたと言える．

表4 解析に要した時間と解析に失敗した回数

被験者	難読化しなかった場合		制御構造を複雑化した場合		処理の順序も複雑化した場合	
A	143	(0)	1065	(0)	3880	(3)
B	87	(0)	672	(0)	1467	(1)
C	132	(0)	778	(1)	3319	(5)
D	192	(1)	912	(0)	2199	(2)
E	297	(0)	413	(0)	2690	(1)
F	195	(0)	236	(0)	1449	(1)
G	118	(0)	891	(1)	4982	(8)
H	74	(0)	400	(0)	1715	(1)
I	207	(0)	548	(0)	4589	(9)
平均	161	(0.11)	657	(0.22)	2921	(3.44)

(括弧内の数値は解析に失敗した回数である．左側の数値は解析に要した時間の秒数である．)

小規模のプログラムを実験に用いた理由は，実験に要する時間の制約によるが，このように短いプログラムの複雑化には限りがある．解析がほとんど不可能な大

規模なプログラムも実在することから, 実際の長いプログラムを難読化する場合には, より大きな効果が得られると予想される.

4.7 まとめと今後の課題

ループを持つプログラムを難読化する 2 通りの方法を提案し，それぞれの方法の有効性を評価する実験について述べた．実験の結果，20 行程度の小規模なプログラムに対してさえも，提案する方法が難読化に有効であることがわかった．

大規模なプログラムは，一般に，元より複雑であることもあり，変数や関数の名前を意味のない文字列に置き換えたり，コメント文を削除したりと言う単純な方法を適用するだけでも解析が困難になる．それらの単純な方法と合わせて, 提案した難読化法を大規模なプログラムに適用して評価実験を行うことは，今後の重要な課題である.

また，機械語プログラムに特化した難読化法について検討することも今後の重要な課題である.

5. おわりに

本論文では，知的財産権の侵害となる代表的な四つの海賊行為p_1〜p_4を抑止する技術について検討した．(p_1) 他人のソフトウェアを無断でコピー，使用する行為．(p_2) 他人のソフトウェアを改変し，無断で再配布，販売する行為．(p_3) 他人の開発したソフトウェアの一部を自分のソフトウェアに組み込み，無断で配布，販売する行為．(p_4) 他人の開発したソフトウェアを解析してアイデアや方式を抽出し，他のソフトウェアに使用する行為．これらの海賊行為を抑止する上では，法的な体制の整備も重要であるが，本論文では，ソフトウェア自身に工夫を施すことにより海賊行為を抑止する，三つのソフトウェア実装法を提案した．

第2章では，p_1の抑止を目的として，従来の鍵付きプログラムを改良した，ネットワーク環境における安全な鍵付きプログラムの実装方法を提案した．提案手法は，従来の鍵付きプログラムでは防ぐことが困難であった，鍵の同時使用，プログラムの改変，といった不正利用をも防止することができる．提案手法は，単にソフトウェアのコピーを抑止するのみでなく，ネットワークを通してユーザにソフトウェアを無料で配布し，代わりにその使用頻度に応じてユーザに課金する「pay per use」への応用が可能である．

第3章では，p_1〜p_3の抑止を目的として，プログラムの実行ファイルに，プログラム開発者の署名やプログラムのユーザIDなどの文字列を電子透かしとして挿入する方法を提案した．提案手法を用いることにより，コピープログラム，盗用プログラム，及び，改変プログラムが不正に再配布された場合に，再配布を行った者を特定したり著作権の所在を明らかにするのに役立つ．提案手法では，プログラム中の電子透かしを自動的に検査できるため，インターネット上のプログラムを自動的に探して透かしを検査し，違法プログラムを発見する「透かし検査ロボット」への応用が可能である．

第4章では，p_2〜p_4の抑止を目的として，ループを持つプログラムを難読化する二通りの方法を提案し，それぞれの方法の有効性を評価する実験について述べた．実験の結果，20行程度の小規模なプログラムに対してさえも，提案する方法が難読化に有効であることがわかった．プログラムを難読化してからユーザに配布することにより，ユーザがプログラムを解析することが困難となるため，プロ

グラムの改変，部品の盗用，アイデアの盗用を抑止することが可能となる．

提案した三つのソフトウェア実装法は，併用して使うことにより多様な海賊行為を抑止することが可能となる．例えば，難読化したプログラムをコンパイルし，電子透かしを挿入してからユーザに配布することにより，p_1〜p_4すべての海賊行為を同時に抑止することができる．さらに，p_1のソフトウェアの違法コピーを強く抑止したい場合は，2章で提案した安全な鍵付きプログラムも同時に併用することが考えられる．

また，提案した三つの方法は，すでに提案されている他の方法と併用することも可能である．例えば，暗号化技術を利用した難改変プログラムの実装法 [3][17] は，プログラムの難読化法との併用が可能である．難読化したプログラムを暗号化し，難改変プログラムとして実装してからユーザに配布することにより，p_2〜p_4の違法行為をより強力に抑止することが可能となる．この難改変技術は，鍵付きプログラムにも応用が可能である．複製の困難なハードウェアキーと難改変プログラムを併用することで，ネットワークに接続されていない計算機環境においても，プログラムが改変されて鍵が無効化されるという不正利用を抑止することが可能となる．難改変プログラムを容易に実装するための技術を開発することは，今後の重要な課題である．

提案手法は，入力データを暗号化した状態のままで計算を行う Mobile Cryptography 技術 [14][39] と併用することも考えられる．この技術は，ネットワーク上の信頼のできない計算機上にプログラムを送り込んで計算を行う場合に，プログラムが解析されたり改造されるのを防ぐことを目的としている．この技術を用いることにより，プログラムを極めて強力に難読化，及び，難改変化できるため，本論文で提案した鍵付きプログラムや，プログラムの難読化法と併用することにより，より安全なプログラムを実装できると期待される．ただし，現状では，Mobile Cryptography 技術は，ごく一部の種類の計算式にしか適用することができないため，今後適用可能な計算式のクラスを広げていくことが重要な課題となる．

本論文で提案した方法のうち，安全な鍵付きプログラム，及び，プログラムに対する電子透かし法は，ネットワーク環境に対応した技術であり，一方，プログラムの難読化法は，ネットワークに非依存の技術である．近年では，普段はネッ

トワークには接続されないが，ISDN 回線等を利用して一時的にインターネットに接続される計算機が増えている．また，いわゆるモバイルコンピュータのように，持ち運び時にネットワークから独立して使用される計算機も普及してきている．今後は，ネットワークに一時的にのみ接続されるような計算機環境においても，効果的に海賊行為を抑止できる技術を開発することが重要な課題となる．

謝辞

本研究の全過程を通して，暖かく見守って下さり，直接の懇切なる御指導を賜ったソフトウェア計画構成学講座 鳥居 宏次 教授に心より感謝致します．

本研究を進めるにあたり，終始有益な御助言と励ましの言葉を頂いた情報科学センター 小山 正樹 教授に心より感謝致します．

本研究に関して，様々な助言，励ましの言葉を頂いた計算機言語学講座 関 浩之 教授に心より感謝致します．

本研究の全課程を通して様々な相談に応じて頂き，懇切丁寧なる御指導，御助言を賜ったソフトウェア計画構成学講座 松本 健一 助教授に心より感謝致します．

本研究を通じて，適切な御助言，御指導を賜った情報科学センター 飯田 元 助教授に心より感謝致します．

本研究を通して，研究者としての基本的な考えの進め方について懇切丁寧に御指導を賜わったソフトウェア計画構成学講座 高田 義広 助手（現オムロン株式会社）に深く感謝致します．

本研究に関して，終始有益な御助言，御指導を頂いたソフトウェア計画構成学講座 島 和之 助手に感謝致します．

本研究に関して終始熱心に相談に応じて頂き，有益な御助言を頂いた認知科学講座 中小路 久美代 助教授に感謝致します．

本研究に関して，特に英語表現において，適切な御助言，御指導を頂いた認知科学講座 高田 眞吾 助手に感謝致します．

本研究に関して，終始熱心に相談に応じて頂き，有益な御助言を頂いた通商産業省工業技術院電子技術総合研究所 一杉 裕志 様に感謝致します．

また，常に温かい御支援を頂き熱心に議論して頂き，本研究の評価実験においても快く御協力を頂いたソフトウェア計画構成学講座の諸氏に感謝致します．

本研究は，ここに書ききれなかった多くの人々の御力添えなくしては決して遂行することができませんでした．直接，間接的に支援して頂いた多くの方々に，心より御礼申上げます．

最後に，これまで私を温かく見守って下さった両親に感謝致します．

参考文献

[1] O. Akashi, K. Moriyasu, and A. Terauchi, "Information distribution by Flea-Market system," *Third International Workshop on Services in Distributed an Networked Environment*, pp.139-146, 1996.

[2] R. B. Anderson, "Proving programs correct," John Wiley & Sons, Inc., 1979.

[3] D. Aucsmith: "Tamper resistant software: An implementation," *Lecture Notes in Computer Science*, No. 1174, Springer-Verlag, pp. 317–333, 1996.

[4] H. Berghel and L. O'Gorman: "Protecting ownership rights through digital watermarking," *IEEE Computer*, Vol. 29, No. 7, pp. 101–103, July 1996.

[5] J. T. Brassil, S. Low, N. F. Maxemchuk and L. O'Gorman: "Electronic marking and identification techniques to discourage document copying," *IEEE Journal on Selected Areas in Communications.*, Vol. 13, No. 8, pp. 1495–1504, Oct. 1995.

[6] E. J. Chikofsky, J. H. Cross II, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, Vol. 7, No. 1, pp. 13–17, Jan. 1990.

[7] C. Cifuentes and A. Fitzgerald, "Copyright in shareware software distributed on the Internet – The trumpet winsock case," *Proceedings of 19th International Conference on Software Engineering*, pp.456–464, May 1997.

[8] 古屋栄男, 松下正, 眞島宏明, 田川幸一, "知って得するソフトウェア特許・著作権 改訂版," アスキー, 1997.

[9] 五代陽子, "初級プロテクト掛け方教室," 情報科学出版社, 1988.

[10] S. Gundavaram, "CGI programming on the World Wide Web," O'Reilly & Associates, Inc., 1996.

[11] J. Hartman, "Understanding natural programs using proper decomposition," *Proceedings of 13th International Conference on Software Engineering*, pp. 62–73, May 1991.

[12] 服部徳秀, 石井直宏, "ソースコードのバリエーション除去システム," 電子情報通信学会論文誌, Vol.J80-D-I, No.1, pp.50–59, Jan. 1997.

[13] 廣瀬直人, 岡本英司, 満保雅浩, "ソフトウェア保護に関する一考察," 1998年暗号と情報セキュリティシンポジウム, SCIS'98-9.2.C, Jan. 1998.

[14] F. Hohl, "An approach to solve the problem of malicious hosts," Universität Stuttgart, Fakultät Informatik, Fakultätsbericht Nr. May 1997.

[15] 一杉裕志, "ソフトウエア電子すかしの挿入法, 攻撃法, 評価法, 実装法," 情報処理学会, 夏のプログラミングシンポジウム報告集, pp.57–64, July 1997.

[16] 飯島邦夫, "ネットワークにおけるソフトウェアのライセンス管理," Interface, 97年9月号, pp. 154–156, Sep. 1997.

[17] 鴨志田昭輝, 松本勉, 井上信吾, "耐タンパーソフトウェアの構成手法に関する考察," 信学技報, ISEC97-59, pp.69–78, Dec. 1997.

[18] 北川隆, 楫勇一, 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'98-9.2.D, Jan. 1998.

[19] 北口秀実, "ソフトウェア特許の理想と現実," 情報処理, Vol. 34, No. 8, pp. 973–982, Aug. 1993.

[20] K. E. Leininger, "The Java developer's tool kit", The McGraw-Hill Companies, Inc., 1997.

[21] Z. Manna, "Mathematical theory of computation," McGraw-Hill, 1974.

[22] G. McGraw and E. Felten, "Java security: hostile applets, holes, and antidotes", John Wiley & Sons, Inc., 1997.

[23] J. Meyer and T. Downing: "JAVA virtual machine", O'Reilly&Associates, Inc., 1997.

[24] 三次衛, 小田久司, "ソフトウェアをめぐる法的問題," 情報処理, Vol. 37, No. 2, pp. 122–127, Feb. 1996.

[25] 松井甲子雄, "デジタル透かし," 画像電子学会誌, Vo.26, No.3, pp.266–274, June 1997.

[26] 門田暁人, 高田義広, 鳥居宏次, "プログラムの難読化法の提案," 情処全大, 分冊 4, pp. 4-263–4-264, Sep. 1995.

[27] 門田暁人, 高田義広, 鳥居宏次, "プログラムの難読化法の実験的評価," 情処学ソフトウェア工学研報, Vol. 96-SE-108-5, pp. 33–40, Mar. 1996.

[28] 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.

[29] 門田暁人, "コピー防止を目的とするプログラムの虫食い実装方式," 夏のプログラミングシンポジウム報告集, pp. 47–54, July 1997.

[30] 門田暁人, 飯田 元, 松本健一, 鳥居宏次, 一杉裕志, "安全な鍵付きプログラムによるソフトウェア保護," ソフトウェアシンポジウム'98 論文集, pp. 108–115, June 1998.

[31] 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'98-9.2.A, Jan. 1998.

[32] A. Monden, "Secure keyed program in network environment," *Proceedings of 20th International Conference on Software Engineering*, Vol. 2, pp. 170–171, Apr. 1998.

[33] R. Mori and M. Kawahara, "Superdistribution: An electronic infrastructure for the economy of future," *Transaction of IPSJ*, Vol. 38, No. 7, pp. 1465–1472, July 1997.

[34] 村山隆徳, 満保雅浩, 岡本栄司, 植松友彦, "ソフトウェアの難読化について," 信学技報, No. ISEC95-25, pp. 9–14, Nov. 1995.

[35] 中野秀男, "インターネットにおけるセキュリティ技術," 第15回ソフトウェア信頼性シンポジウム論文集, pp. 2–11, Dec. 1994.

[36] 中澤良充, "CD-ROMによるソフトウェア流通技術," 信学技報, ISEC94-18, pp. 41–46, Sep. 1994.

[37] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, Vol. 21, No. 1, pp. 19–31, Jan. 1995.

[38] P. Samuelson, "Reverse-engineering someone else's software: is it legal ?," *IEEE Software*, Vol. 7, No. 1, pp. 90–96, Jan. 1990.

[39] T. Sander, C. F. Tschudin, "Towards mobile cryptography," *Proceedings of Security & Privacy '98*, May 1998.

[40] 鳥居直哉, 木島裕二, 勝山恒男, 小森宰次, "超流通のためのシステム開発," 信学技報, ISEC94-22, pp. 59-66, Sep. 1994.

[41] "特集インターネット違法ソフト," ゲームラボ, 1997年6月号, pp. 39–70, June 1997.

[42] 山本哲也, 渡辺創, 嵩忠雄, "すべての結託ユーザを特定可能な画像電子透かし法," 1998年暗号と情報セキュリティシンポジウム, SCIS'98-10.2.B, Jan. 1998.

[43] Hashjava home page.
<http://www.blackdown.org/~kbs/ hashjava.html>.

[44] Business Software Alliance home page, <http:// www.bsa.org>.

[45] Software Publishers Association home page, <http: //www.spa.org>.

付録

電子透かし挿入のための JAVA バイトコード変換規則

変換前の opcode	変換後の opcode	挿入されるビット
iadd	iadd	000
isub	isub	001
imul	imul	010
idiv	idiv	011
irem	irem	100
iand	iand	101
ior	ior	110
ixor	ixor	111
ishl	ishl	0
ishr	ishr	1
iushr	—	–
ladd	ladd	000
lsub	lsub	001
lmul	lmul	010
ldiv	ldiv	011
lrem	lrem	100
land	land	101
lor	lor	110
lxor	lxor	111
lshl	lshl	0
lshr	lshr	1
lushr	—	–

変換前の opcode	変換後の opcode	挿入されるビット
fadd	fadd	00
fsub	fsub	01
fmul	fmul	10
fdiv	fdiv	11
frem	—	–
dadd	dadd	00
dsub	dsub	01
dmul	dmul	10
ddiv	ddiv	11
drem	—	–
ifeq	ifeq	0
ifeq	ifne	1
iflt	iflt	00
ifge	ifge	01
ifgt	ifgt	10
ifle	ifle	11
if_icmpeq	if_icmpeq	0
if_icmpne	if_icmpne	1
if_icmplt	if_icmplt	00
if_icmpge	if_icmpge	01
if_icmpgt	if_icmpgt	10
if_icmple	if_icmple	11
iconst_m1	iconst_m1	00
iconst_0	iconst_0	01
iconst_1	iconst_1	10
iconst_2	iconst_2	11

変換前の opcode	変換後の opcode	挿入されるビット
iconst_3 iconst_4 iconst_5	iconst_3	0
	iconst_4	1
	—	–
lconst_0 lconst_1	lconst_0	0
	lconst_1	1
fconst_0 fconst_1 fconst_2	fconst_0	0
	fconst_1	1
	—	–
dconst_0 dconst_1	dconst_0	0
	dconst_1	1
ifnull ifnonnull	ifnull	0
	ifnonnull	1
iinc x	iinc x'	x の部分に 8bit
wide iinc x	wide iinc x'	x の部分に 16bit
bipush x	bipush x'	x の部分に 8bit
sipush x	sipush x'	x の部分に 16bit