ソフトウェア工学 108－5
（1996．3．22）

プログラムの難読化法の実験的評価

門田 暁人, 高田 義広, 鳥居 宏次

奈良先端科学技術大学院大学 情報科学研究科

〒630-01 奈良県 生駒市 高山町 8916 番地の 5 (高田)

あらまし　ソフトウェアの保守, 再利用などのためには, 解析や理解が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, 完成したプログラムを多数のユーザに配布する場合には, システムの安全性の確保や知的財産権の保護などのために, 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. そのような場合には, 解析が容易なように作成したプログラムを, 解析が困難になるように変換する方式が有効であると考えられる. 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. 本発表では, ループを含むプログラムを自動的に難読化する 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ために行った実験について報告する. 実験の結果, 極めて小規模なプログラムに対しても, 提案する方法が有効であることがわかった.

和文キーワード　プログラムの難読化, プログラム理解, 知的財産権, 安全性, ループ不変式, リバースエンジニアリング

An Experiment to Evaluate Methods for Program Scrambling

Akito MONDEN, Yoshihiro TAKADA, and Koji TORII

Graduate School of Information Science, Nara Institute of Science and Technology

8916-5 Takayama, Ikoma-Shi, Nara 630-01 (c/o Takada)

Abstract

It is important to write programs that are easy to analyze or understand so that the load on software maintenance and reuse can be lessened. On the other hand, when finished programs are released to many users, there are cases where programs should not be analyzed easily, in order to protect the intellectual properties or to insure system security. In such cases, we believe that it is effective to write programs that are easy to analyze at first, and transform it so that it becomes difficult to analyze. Such an equivalent transformation is called "program scrambling." We present two methods for automatically scrambling programs that contain loops, and report on an experiment for validation of the methods. The result of the experiment shows that the methods are effective even for very small programs.

英文 key words　program scrambling, program understanding, intellectual property, security, loop invariant, reverse engineering

1 はじめに

ソフトウェアの保守, 再利用などのためには, 解析や理解が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, 2 章で述べるように, 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 3 者に漏洩することを防止したい場合である. 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 1 のような方式が有効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, 完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. 以上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 [5].

難読化の対象は, 機械語のプログラムとその他の言語で書かれたソースプログラムとの両方である. ユーザには, 機械語プログラムだけでなく, ソースプログラムが配布される場合も多いからである. 例えば, フリーウェアの多くは, 多くの種類の計算機で利用できるように, ソースプログラムの形式で配布されている. また, インタプリタ言語で書かれたプログラムの多くも, ソースプログラムの形式で配布されている. また, 機械語プログラムから, ソースプログラムが容易に復元されることもある. そのようなソフトウェアの復元の技術は, リバース・エンジニアリングと呼ばれている.

本発表では, 高級言語で書かれ, かつ, 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. そして, 方法の有効性を評価するために行った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 [6] 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, 命令を並び変えたりする.

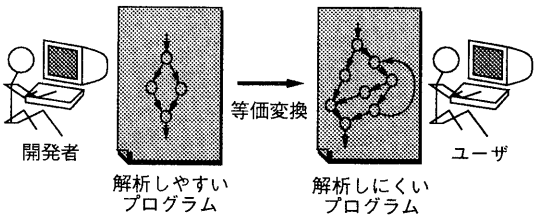


図 1: プログラムの難読化

提案する難読化法は, C 言語などのソースプログラムを対象としており, プログラム中のループの部分を複雑に変換する方法である.

2 難読化の有用な場合

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.

2.1 システムに対する不正を防止したい場合

プログラムが組み込まれているシステムに対して, 悪意を持つユーザや第 3 者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減らすことができると言える.

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 [7] 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, そのような不正の方法が発見される危険性が増すことがある.

2.2 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. 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 [2, 4]. 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.

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.

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. つまり, 解析とは, プログラムよりも, むしろ, そのような知識を対象とする行為と言える. 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.

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. 仮に, プログラムを暗号化したとすれば, 解析は困難になるが, そのままではプログラムを実行できない. 暗号を解読するための鍵と組にしてプログラムを配布すれば, 復号して実行できるが, 復号後のプログラムを解析されるかもしれない. 難読化されたプログラムは, 変換前と同様に実行可能な形式であり, 変換前と同一の仕様を満たす必要がある.

プログラムを変換すると, 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, 計算機の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考えられる. しかし, 時間的な実行効率の低下は, 許容されると言えない. 計算機の速度も増加しているが, 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, 定義中で明確な基準を述べていない. 実用上の観点からは, おそらく, P に対する P' の大きさが2, 3倍程度であれば許容され, 10倍程度であれば支障があると思われる.

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, C 言語, C++, Pascal などで書かれ, for 文, while 文, repeat 文などを含むプログラムである. 図2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

```
int sum (int N, int A[]) {
    int x, i;
    x = 0;
    for(i = 0; /*繰り返しの開始*/ i < N; i++) {
        if (A[i] > x) x = A[i];
    }
    return x;
}
```

図 2: ループを含むプログラム

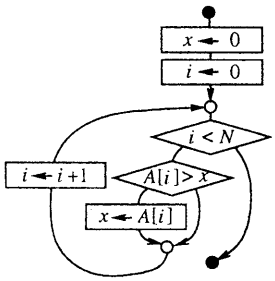


図 3: フローチャート

ば, 図2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.

一般に, 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, 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 [1, 3]. 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. 従って, ループを含むプログラムを難読化するには, そのループ不変式の発見を困難にすればよいと考えられる. 図2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.

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.

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2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3に示すようなフローチャートを使用する. この図は, 図2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, プログラムの開始点と終了点 (塗りつぶした小円), 分岐文や条件文の他の実行文を表す節 (矩形), 条件分岐を表す分岐点 (菱形), 分岐点に対する合流点 (小円) である. なお, 分岐点以外の 1 個の節から複数の辺が出ることは許されないし, 合流点以外の 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.

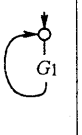
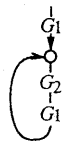
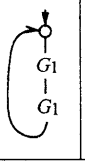
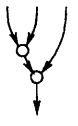
5.1.1 変換の手順

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) グラフ上の節や辺に対して, 表 1に示す 4 個の変換規則を無作為に何度も適用する.
(3) 3 個以上の辺が入っている合流点が残っていないことを調べる. 残っていれば, 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, 有向グラフ中の特定の部分グラフに適用され, それを別の部分グラフに置き換える. 表中では, 各規則を, 変換前と変換後の部分グラフの組で表している. 各 G_i は, 0 個以上の節か

表 1: 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				
変換後				

らなる任意のグラフである. ただし, どの G_i についても, 1 個の辺が入り, 別の 1 個の辺が出ている. G_i へ入っている辺は, G_i が空でない, つまり, 1 個以上の節を持つならば, G_i 中の節へ入っていることを表す. G_i から出ている辺は, G_i 中の節から出ていることを表す. G_i が空である, つまり, 1 個の節も持たないならば, 両辺は同一の辺を表す.

各規則は, 表中の変換前のグラフの各節について, 表中に明示されていない辺が入っていたり出ていたりしても適用できる. ただし, 第 1, 第 2, 第 4 規則においては, 節の複製が起こるので, それらの節に入っていたり, それらの節から出ていたりする辺については, 注意が必要である.

明示されていない辺の入っている合流点を複製した場合は, 複製された合流点の一方を適当に選んで, その辺を入れるようにする. ただし, 複製前の合流点へ, 明示されていない辺を含めて 2 個の辺しか入っていない場合には, 複製後の一方の合流点へ, 1 個の辺しか入っていないことになる. そのような合流点は必要ないので, 複製後に直ちに削除する.

明示されていない辺が出ている分岐点を複製した場合は, その辺も複製する. ただし, その辺の入っている節が合流点でない場合には, 新たに合流点を挿入して, 複製した両辺がその合流点に入るようにする.

```
int sum (int N, int A[]) {
    int x, i;
    x = 0;
    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 4: 制御構造を複雑化したプログラム

図 4は, 図 2のプログラムに本方法を適用して得られた例である.

5.1.2 仕様の保存

本方法を構成する 4 段階の内, 第 1, 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 1の単純な変換の繰り返しであるが, 各変換はプログラムの仕様を変えない. 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れる.

5.1.3 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, 本方法を適用しても, プログラム実行効率は保存される.

5.1.4 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.

5.1.5 自動化

本方法を構成する4段階は, どれも自動化が可能な程度に単純である. 従って, 本方法の全体についても, ソフトウェアツールによる自動化が可能である.

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2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, その処理が, 例えば, 0, 5, 1, 4, 3, 2 の順に実行されるように制御構造を変える.

ただし, 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ることが多い. 従って, 実行効率を全く低下させずに, そのよう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, ある特徴を持つループに対して, 効率を低下させないような変換を予め調査する. 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.

5.2.1 変換の手順

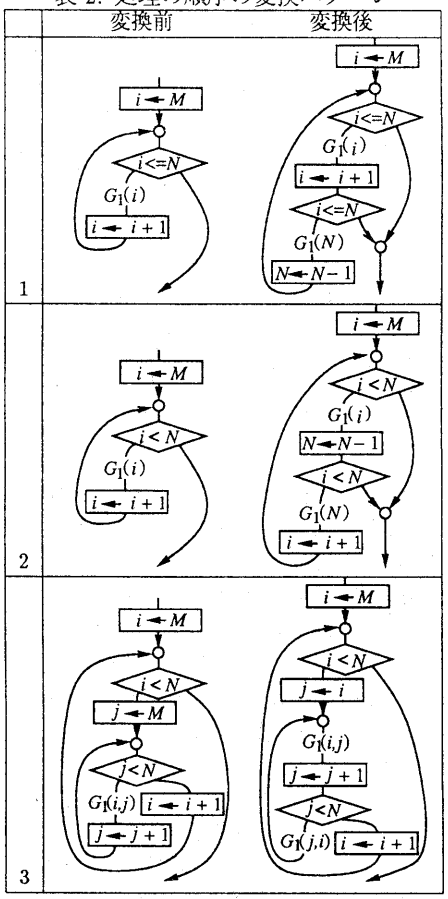
具体的な変換の手順は, 次の通りである.

(1) プログラムの静的解析によって, 繰り返しの回数が固定されていて, かつ, 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検出する.

(2) そのようなループに対して, 適用可能な変換パターンをカタログから選出し適用する. 複数のパターンが適用可能な場合には, 無作為に選出する.

(3) 以上の操作を適当に繰り返す.

表 2: 処理の順序の変換パターン



これまでに筆者らの得た変換パターンを表2に例示する. 表中では, 各変換パターンを, やはり, 変換前と変換後の部分グラフの組で表している. $G_n(i)$ は, 繰り返される回数が固定されていて, かつ, 順

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

図 5: 処理の順序を複雑化したプログラム

序を入れ換えてもよい処理を表す部分グラフである. i は, $G_n(i)$ 中で参照される制御変数を表す.

図5は, 図2のプログラムにパターン2を適用して得られた例である.

5.2.2 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.

5.2.3 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2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.

5.2.4 大きさの増加

表2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. ただし, 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.

5.2.5 前方法との併用

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. ただし, 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.

5.2.6 自動化

変換の手順における第2段階は, 自動化が可能であると考えられるが, 第1段階は, 必ずしも自動化できると言えない.

第1段階では, プログラムの静的解析により, 各ループの繰り返しの回数が固定されているか, そして, 繰り返される処理の順序を入れ換えてもよいかを判定する. 繰り返しの回数の判定は, 比較的容易であり, 自動化できると考えられる. ところが, 処理の順序の判定は, 容易でない. 単純な分析を行うことにすれば, 自動化は可能になる. ただし, 処理の順序を入れ換えられると検出できるループが少なくなり, その結果, 変換パターンを適用できる率が低くなる. 逆に, 高度な分析を行うことにすれば, 変換パターンを適用できる率が高くなるが, 自動化が困難になる. どの程度の分析を行うべきかについては, 現在も研究中である.

6 方法の評価実験

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, 予め, 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.

被験者に対するプログラムの割り当てについては, 特に次の 2 点に配慮した. 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. 第 2 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.

6.2 実験の結果

表 3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. 難読化したプログラムは, 難読化していないプログラムよりも解析が困難であったと言える. また, 制御構造を複雑化するだけでなく, 処理の順序を複雑化する方が, 解析が困難になったと言える.

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. やはり, 種類 2 のプログラムの解析が最も困難であったと言える.

表 3: 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.)

7 おわりに

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ための実験について報告した. 実験の結果, 20 行程度の極めて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については, 一般に, プログラムの一部の解析が困難であるだけで, 全体の解析が困難になる場合がある. 従って, 提案した方法を大規模なプログラムに適用すれば, 報告した実験の結果よりも大きな効果が得られると予想される. 大規模なプログラムに対する評価実験を行うことは, 今後の重要な課題である.

参考文献

[1] Anderson R. B. 著, 有澤 誠 訳: "演習 プログラムの証明", 近代科学社 (1980).

[2] 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).

[3] Manna Z. 著, 五十嵐 滋 訳: "プログラムの理論", 日本コンピュータ協会 (1975).

[4] 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).

[5] 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, pp. 4-263–4-264 (Sep. 1995).

[6] 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, ISEC95-25, pp. 9–14 (Nov. 1995).

[7] 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 (Dec. 1994).