プログラムの難読化法の提案

5G－7

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

1　はじめに

ソフトウェアの修正, 保守, 再利用などのためには, 理解の容易なプログラムを作成することが重要である. ところが, 逆に, プログラムの理解を困難にすることが要求される場合がある. それは, プログラム中で使用されているアイデアや方式が, プログラムを解析されることによって, 不特定の者に洩れることを防止したい場合である.

理解を困難にすることと容易にすることとは, 一見, 矛盾するように思われるが, 次の方式により両立できると考えられる. まず, 理解が容易になるようにプログラムを作成し, 次に, 理解が困難になるようにそれを変換する. 変換においては, プログラムの機能や性能が変わらないよう配慮する. そして, 変換前のプログラムは, 保守や再利用のために厳重に保管しておき, 変換後のプログラムだけを使用したり配布したり公開したりする. このような目的で行なわれるプログラムの変換を, プログラムの難読化と呼ぶことにする.

本発表では, まず, 難読化が有用な場合について論じ, その結果に基づいて難読化のより厳密な定義を与える. そして, 難読化を行なう方法の例を示し, それらの方法の評価実験について述べる.

2　難読化の有用な場合

プログラムが解析されることによりアイデアや方式が知られることの防止は, 主に次の2つの場合に要求されると考えられる.

1つは, ソフトウェアの知的財産権を保護したい場合である. そのアイデアや方式が独創的で価値が高い場合, もし, それが一般に知れ渡り使用されるようになると, その価値が失われることになる.

もう1つは, システムの安全性 (security) を確保したい場合である. そのプログラムがあるシステムに組み込まれていて, そのアイデアや方式がシステムへの不正操作の防止に関係している場合, もし, システムのユーザにそれが洩れると, 不正操作を許す危険性が増すことになる.

なお, ここで言うプログラムは, 機械語プログラムだけと限らない. 機械語プログラムからソースプログラムを導出するリバースエンジニアリングの技術も研究されているし, ソースプログラムで配布されるプログラムもあるからである.

3　難読化の定義

前章で述べたように, 理解を困難にする対象は, 厳密に言うと, プログラム自身ではなく, プログラム中で使用されているアイデアや方式である. そこで, “理解する” と言う行動を次のように定義する.

定義1　プログラム P と P に関する命題 Q とに対して, Q を論理的に証明することを, P に関して Q を**理解**すると言う.

これに基づいて, 難読化を次のように定義する.

定義2　P と Q とに対して, 次の3条件を満たすプログラム P' を求めることを, Q に関して P を**難読化**すると言う.
(1) 任意の入力について, P と同一の出力を返す.
(2) 実行時の時間的効率が P と同等である.
(3) Q の理解に P に対してよりも時間がかかる.

4　難読化の方法

前述の定義に基づいて, 2通りの方法を提案する. どちらの方法も, ループを持つプログラムのクラスを対象とする. そして, プログラム出力についての任意の命題に関して難読化を行なう. そのようなプログラムの理解においては, ループ不変式 [1] を見つけ

A Method for Program Scrambling
Akito Monden, Yoshihiro Takada, and Koji Torii
Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5 Takayama-cho, Ikoma-shi, Nara 630-01, Japan

ることが重要であるが, どちらの方法もそれを困難にする.

方法 1 制御の流れの複雑化 実行される処理の順序は全く変えずに, 分岐や繰り返しの構造だけを複雑に変える.

方法 2 処理の順序の複雑化 入れ換えてもよい処理を見つけて, 処理の順序を複雑に変える.

どちらの方法も, いくつかの変更規則を無作為に適用することにより行なえるので, 自動化が可能である.

図 1 の C 言語のプログラムに両方法を適用した結果を図 2, 3 に示す. 分析するとわかるように, 3 プログラムは, 任意の入力に対して, 同一の出力を返すし, 実行される比較・加減算の回数も同一である.

```
x = A[0][0];
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        if (A[i][j] > x) x = A[i][j]
    }
}
```

図 1: 2 次元配列中の最大値を求めるプログラム

```
x = A[0][0];
i = j = 0;
for (;;) {
    if (j >= M) {
        i++;
        if (i >= N) break;
        j = 0;
    }
    if (A[i][j] > x) x = A[i][j]
    j++;
}
```

図 2: 方法 1 を適用した結果

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

図 3: 方法 2 を適用した結果

5 方法の評価実験

5.1 実験方法

与えられたプログラムと命題についての理解を, 12 名の被験者に, のべ 36 回, 試みてもらった.

用意したプログラムは, 難読化していない極短い 3 個と, それらに方法 1, 2 を適用して得られる 6 個との, 合わせて 9 個である. (図 1〜3 のプログラムを含む.) そして, 各プログラムに対して, "X を入力すると Y を出力する" と言う形式の命題を用意した.

各被験者には, 予め, 命題の証明を記述する方法を指導し, 2 回以上の練習を行なってもらった. そして, 9 個のプログラムの中から, 難読化していないものと方法 1, 2 を適用したものとの合わせて 3 個を割り当てて, それぞれ命題の証明を記述してもらった. そして, 証明を記述し終るまでの時間を, 理解に要した時間として測定した. プログラムの割り当て方や試行の順序などについては, 偏りが生じないよう十分に配慮した.

5.2 実験結果

難読化しなかったプログラム (無変更), 方法 1 を適用したプログラム (方法 1), 方法 2 を適用したプログラム (方法 2) に分けて, 理解に要した時間を図 4 に示す. グループ内の平均は 445 秒, 603 秒, 759 秒であり, その比率は 1 : 1.35 : 1.71 であった. 分散分析の結果, $F = 5.73 > F_{0.99}(2, 33)$ となり, 3 グループの平均の同一性は 1%の危険率で棄却された. つまり, 方法 1, 2 は共に有効であり, 更に, 方法 2 は方法 1 よりも有効であると言える.

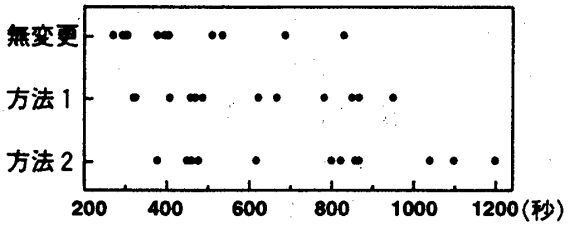

図 4: 理解に要した時間

6 おわりに

極短いプログラムに対しても, 自動化できる程の簡単な方法が難読化に有効であることがわかった. 今後は, 大規模なプログラムに対するより高度な方法について検討する予定である.

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).