耐逆コンパイル難読化手法の提案
―種々のエラーを引き起こす手法―

M2007MM020 永井義昭
指導教員：真野芳久

1　はじめに

近年、オブジェクトコードからソースコードに変換する逆コンパイルが問題となっている。特に、Java クラスファイルは逆コンパイルされやすい構造をしているので、ソースコードを解析され、リバースエンジニアリングや改ざんといった攻撃の脅威となっている。これを防ぐ技術にプログラムの機能を保ったまま ソースコードを読みづらいものに変換する難読化がある。今までに Java を対象とした数多くの難読化手法が提案されてきたが、その中でも逆コンパイルを防ぐ観点からの難読化手法の系統的な研究は十分なされていない。

本研究では現在様々な分野で活用されている Java クラスファイルを対象とし、逆コンパイルを防ぐ難読化の検討、分類、基礎の部分の関連研究の調査を行った。そして、逆コンパイルを行うと種々の要因からコンパイルエラー、実行エラーを引き起こす新しい難読化手法を提案する。さらに、その提案手法が逆コンパイルを防ぐ能力について既存の逆コンパイラと本研究で作成した実験システムを用いて実験的に分析を行う。

2　難読化と耐逆コンパイル難読化

2.1　難読化

[1] を参考に、難読化の定義を次に定める。

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 2 つの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。
- (解析困難さの増加) P' は Q に関する解析に P よりも時間がかかる。

2.2　従来の難読化のモデル

Collberg らが分類した従来の難読化 [2] はプログラムを読みづらいものとすることで攻撃者にプログラムを解析することにかかる労力をプログラムの価値と比べ高くすることで事実上解析を不可能にすることが目的である。この従来の難読化のモデルは図 1 のようになり、攻撃者は難読化が施されたクラスファイルから逆コンパイラを使いソースコードを得てそのソースコードを解析し、プログラムに改竄を行ったり、部分的に盗用するためにプログラムの解析を行う。

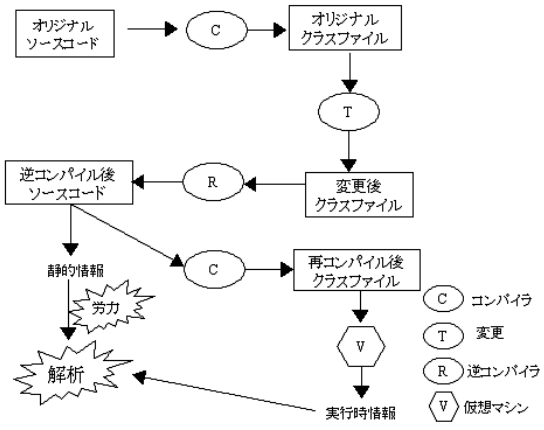

図 1　従来の難読化のモデル

2.3　耐逆コンパイル難読化

本研究で提案する耐逆コンパイル難読化とは、難読化する対象を Java クラスファイルとし、2.1 節で定義した難読化の定義を満たし、なおかつ以下に示す 3 つの条件のうち、少なくとも 1 つを満たすものと定義する。

- 出力失敗…逆コンパイラがエラーなどによりソースコードの生成に失敗する
- コンパイルエラー…ソースコードの記述形式が不正であるため、コンパイル時にエラーが起きる
- 実行エラー…再コンパイル後クラスファイルの実行をするとエラーが起きる

本研究では耐逆コンパイル難読化を上記の 3 つに分類し、さらにコンパイルエラーは以下の 3 つ

- 字句エラー…字句解析の時点でエラーが起きる
- 構文エラー…構文解析の時点でエラーが起きる
- 文脈エラー…意味解析の時点でエラーが起きる

に分類し、実行エラーは以下の 2 つ

- 実行停止…実行時エラーによってプログラムが実行停止する
- 論理エラー…実行結果がオリジナルクラスファイルのものと異なる

に分類する。

それぞれのエラーを起こす難読化手法の関連研究として字句エラーに識別子をソースコード上で不正なものに変える Illegal identifiers[3]、構文エラーに代入文の左辺と右辺の間にスタックに値の積み出しを行う Un-letting Completion of Statement[4]、文脈エラーにクラスとネストされたクラスに同じ名前を使う Nested TypeNames[3]、論理エラーにメソッド名にできるだけ同じ名前を使う Overloading Unrelated Methods がある。

この節で分類した耐逆コンパイル難読化のモデルを図 2 に示す。

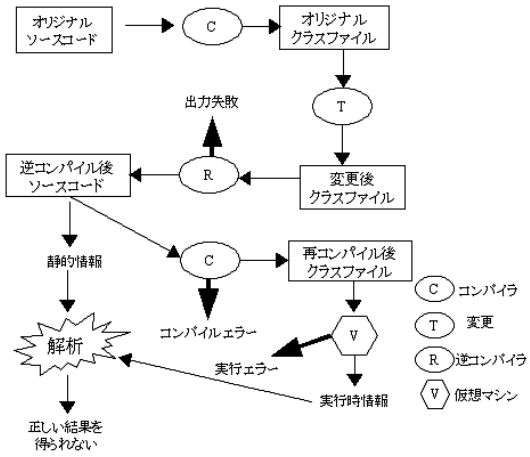

図 2 耐逆コンパイル難読化のモデル

本研究で提案する耐逆コンパイル難読化は逆コンパイルされたコードをそのままコンパイル、実行することができないようすることで、攻撃者に対しソースコードでのエラー要因の除去、クラスファイルレベルでの解析をさせることによって解析の時間と労力をかけさせることを目的としている。また、耐逆コンパイル難読化は逆コンパイルするとエラーが出るプログラムを解析する価値があるかという疑問および、エラー箇所を正しく除去できたかという証明の難しさを攻撃者に与える。

3 提案手法

3.1 コンパイルエラーを引き起こす手法

この節では逆コンパイル後したソースコードに文脈エラーを出す手法についていくつか記述する。基本的なアイディアはソースコードとクラスファイルの形式の制約の違いを利用することで逆コンパイル後ソースコードにコンパイルエラーを引き起こすというものである。図 3 にコンパイルエラーを引き起こす耐逆コンパイル難読化手法のモデルを示す。

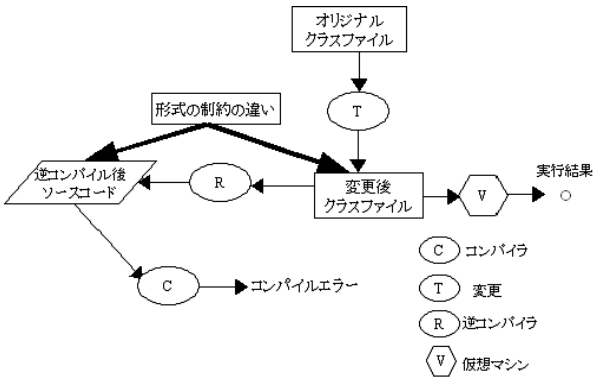

図 3 コンパイルエラーを引き起こす手法のモデル

3.1.1 チェック例外を利用した手法

例外クラスには例外処理を記述しないとコンパイルエラーを起こすチェック例外と、例外処理が任意である非チェック例外がある。Error と RuntimeException クラスおよびそれらを継承するものは非チェック例外であり、それ以外のものはチェック例外である。チェック例外の例として、1 行ずつ読み込む BufferedReader.readLine() の記述には IOException クラスで例外が発生し、スロー宣言か catch 節に例外処理をしなければコンパイルエラーとなる。このチェック例外はコンパイル時にはソースコード上で例外処理の記述をしなければエラーとなるが、バイトコード上では例外処理の記述をしなくてもエラーが出ず、実行することができる。そこで、実際には実行しない偽の条件文を用意し、そこにバイトコードで、チェック例外が必要な記述をすることで、逆コンパイル後コードにコンパイルエラーを引き起こす。図 4 にこの手法で難読化したクラスファイルを Jad によって逆コンパイルした結果を示す。

```
Class C{C(){a=1;b=1;}
  void M(){
    if(a * a == 7 * b * b − 1){
BufferedReader bufferedreader=new bufferedRea
der(newInputStreamReader(System.in));
  String s = bufferedreader.readLine();}}
  int a; int b;}
```

図 4 チェック例外手法の逆コンパイル結果

図 4 で「a*a==7*b*b − 1」という式は a と b にどんな値が入っても必ず偽となるため if 文の中は実行されないが、チェック例外の記述があり、例外処理も行っていないのでコンパイルエラーが起きる。

3.1.2 アクセス修飾子を利用した手法

インタフェースを実装したクラスで public 修飾子を付けないでインタフェースの抽象メソッドを実装するとコンパイルエラーとなる。そこで、インタフェースを実装しているクラスとインタフェースに実際には実行しないメソッドを追加することで逆コンパイルコードにコンパイルエラーを起こすことができる。図 5 にこの手法で難読化したクラスファイルを Jad によって逆コンパイルした結果を示す。

```
class C1 implements I{
  C1(){}  void M(){}}
interface I{public abstract void M();}
```

図 5 アクセス修飾子手法の逆コンパイル結果

図 5 ではインタフェース I を実装したクラス C1 はメソッド M を実装しているが、public 修飾子がないためコンパイルエラーが起こる。

3.1.3 継承関係を利用した手法

抽象クラスを継承して、非抽象クラスをつくる場合には、継承元である抽象クラスの抽象メソッドをすべて実装しなければコンパイルエラーとなる。そこで、オリジ

ナルのプログラムに抽象メソッドを持つ抽象クラス C1 と親クラスを持たない非抽象クラス C2 があった時、C2 が C1 を継承するようにすることで逆コンパイルコードにコンパイルエラーを起こすことができる。図 6 にこの手法で難読化したクラスファイルを Jad によって逆コンパイルした結果を示す。

```
abstract class C1{
    C1(){}
    public abstract void M();}
class C2 extends C1{C2(){super();}}
```

図 6 継承関係を利用した手法の逆コンパイル結果

図 6 ではクラス C2 は抽象クラス C1 を継承しているが、抽象メソッド M を実装していないため、コンパイルエラーとなる。

3.1.4 到達不能な例外処理による手法

try 文の catch 節のパラメータにある例外クラス A を記述した後、次の catch 節のパラメータに A のサブクラス B を指定すると B の catch 節は到達不能となり、コンパイルエラーとなる。そこで、オリジナルの catch 節のパラメータにサブクラスを持つ例外クラスの記述があった時、その後にそのサブクラスの例外処理をパラメータとする catch 節を追加することで逆コンパイルコードにコンパイルエラーを起こすことができる。図 7 にこの手法で難読化したクラスファイルを Jad によって逆コンパイルした結果を示す。

```
try{java.net.URL url=new java.net.URL("http:
//javafaq.jp/S172.html");
  java.io.InputStream is = url.openStream();}
catch(IOException ie){ie.printStackTrace();}
catch(MalformedURLException me){
  me.printStackTrace();}
```

図 7 到達不能例外処理手法の逆コンパイル結果

図 7 では IOException の例外クラスの後にそのサブクラスである MalformedURLException が記述されているのでこの catch 節は到達不能となりコンパイルエラーが起きる。

3.2 実行エラーを引き起こす提案手法

この節では実行エラーを引き起こす提案手法をいくつか記述する。基本的なアイディアはオリジナルクラスファイルと再コンパイル後クラスファイルの間の何らかの違いに着目し、その違いを利用してオリジナルクラスファイルに仕掛けを加えるというものである。実行エラーのモデルは図 8 になる。

3.2.1 実行時間の違いを利用した手法

JODE では load 命令の後に pop 命令を入れて無駄なコードを挿入すると、逆コンパイル後ソースコードに if(i!=0)/*empty*/のような無駄なコードを出力する。こ

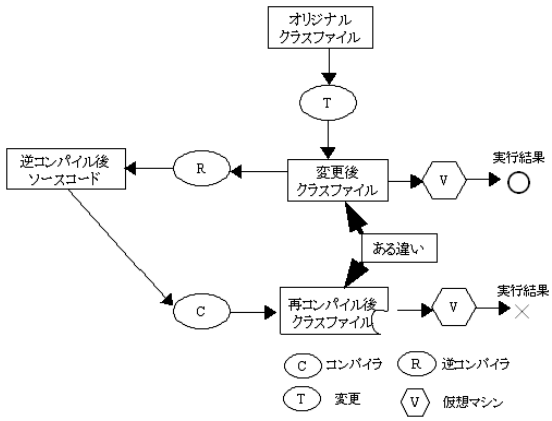

図 8 実行エラーを引き起こす手法のモデル

の無駄なコードは、デッドコードなので無視されるはずであるが、

```
for(int i=0; i<100*1000*1000; i++)
    if (i != 0) /* empty */
```

上のような形を取る場合、繰り返しの中が空文の場合と比べると実行時間が増加する。この実行時間の違いを利用して論理エラーを引き起こす手法を適用し、JODE による逆コンパイルした結果を図 9 に示す。

```
long l=System.nanoTime();
for(int i=0;i<10000000;i++){/* empty */}
    long l_0_=System.nanoTime();
    for(int i=0;i<10000000;i++){
        if(i!=0){/* empty */}}
    long l_1_= System.nanoTime();
if(9L*(l_0_-l)<7L*(l_1_-l_0_))l_1_=l_0_/0L;
```

図 9 実行時間違いの手法の逆コンパイル結果

図 9 では空の for 文と無駄なコードが入っているコードの実行時間を System.nanoTime() で計測し、if 文で無駄なコードが入った for 文の方が実行時間がかかっていれば 0 の除算をし、実行時にエラーを起こすようにしている。System.nanoTime の精度はプラットホームに依存するためプラットフォームごとに実行時間の違いを相対的に決定する必要がある。

3.2.2 不正な識別子を利用した手法

[3] で、識別子を不正なものに変えることで逆コンパイル後ソースコードにコンパイルエラーを起こす illegal identifiers という手法があるが、この手法は逆コンパイラによっては逆コンパイル時に正しい識別子に自動的に変換されることがあり、逆難読化ツール [5] も作成されている。

本研究ではこの illegal identifiers の手法を使って論理エラーおよび実行停止を引き起こす手法を提案する。その方法は攻撃者あるいは逆コンパイラが不正な識別子を正しい識別子に改名することで発生するオリジナルクラスファイルと再コンパイル後クラスファイルの違いを利

用し、Java のリフレクションを使い実行時に識別子の名前を取得する処理とその名前を条件式とする if 文をクラスファイルに追加する。図 10 にこの手法を適用し、Jad によって逆コンパイルした結果を示す。

```
class C{
public static void main(String arg0[])throws
Throwable{
Class class1 = Class.forName("C");
Field afield[] = class1.getDeclaredFields();
 String s = afield[0].getName();
        if(s == "33")_fld33++;
        System.out.println(_fld33);
  }static int _fld33 = 0;
```

図 10 不正な識別子手法の逆コンパイル結果

クラスファイルに対する変更として、フィールド変数名を ”33 ”のようにソースコード上で不正なものに変え、リフレクションによって実行時にフィールド変数名を取得する処理と取得した変数名と変更前の変数名の比較を条件式とする if 文の追加を行っている。if 文の中身は本来実行するコードであるが、図 10 のプログラムでは if 文の中身は実行されないため、論理エラーが起きる。

4 実験

4.1 実験環境

本研究で作成した実験システムにはクラスファイル変換ライブラリとして Javassist (http://www.csg.is.titech.ac.jp/~chiba/javassist/) を使いプログラムの作成を行った。また、難読化ツールを使った提案手法のクラスファイルへの適用は自動で行えるが、実験結果を得るのに CCK (http://bcel.sourceforge.net/cck.html) を使って手動で埋め込みを行った部分もある。使用した OS は WindowsXP で CPU は Intel(R)Core(TM) T7200 2.00GHz、メモリは 512MB である。Java のバージョンは 1.6.0_11 である。

4.2 実験結果

実験データはオリジナルクラスファイルと変換後クラスファイルとの間のファイルサイズ増加量、コード追加部分を繰り返し実行し、1 回あたりの平均実行時間の増加量を測った。実行時間の測定は手法を施した箇所を System.nanoTime() を使って測定した。なお、実行時間の変化がないものについては省略した。

表 1 の結果から 3.1.2 節、3.1.3 節、3.1.4 節の 3 つの手法については実行時間の増加はなかった。3.1.1 節の手法については実行時間の増加は if 文の比較で起こったものであると考えられる。リフレクションを使う 3.2.2 節の手法は実行時間の増加は比較的多く、実行時間の違いを出すために 107 回の for 文を 2 つ追加する 3.2.1 節の手法は実行時間が最も大きくなった。

Jad(http://www.kpdus.com/jad.html)、JODE(http://jode.sourceforge.net/)、Jdec(http://jdec.sourceforge.net/)、SourceAgain(http://www.ahpah.com/product.htm) の 4 つの逆コンパイラで提案手法を逆コンパイルした結果は 3.2.1 節の手法は JODE に対してのみ有効であることを除き、逆コンパイル後ソースコードにコンパイルエラーが出たり、実行結果に違いが生じた。この結果、提案手法の既存の逆コンパイラに対する耐性はあるということが分かった。

表 1 実験結果

手法	3.1.1	3.1.2	3.1.3	3.1.4	3.2.1	3.2.2
ファイル増加量 (バイト)	414	59	13	129	169	315
1 回あたりの平均実行時間増加量 (ナノ秒)	3.63	-	-	-	13.7×10^6	1731.5

5 おわりに

耐逆コンパイル難読化手法の提案と実験を行った。耐逆コンパイル手法は攻撃者が再コンパイル後クラスファイルから動的情報を得ることを防ぐことによって解析を困難にすることができるが、例えば重要なアルゴリズム部分など、部分的な静的解析を困難にすることはできない。そのため、より解析を困難にさせるため、従来のプログラムを読みづらくする難読化と組み合わせて使う必要がある。

提案手法の耐性の検討については既存の逆コンパイラに対してのみで難読化部分を削除したり、エラーがでないようにプログラムに変更を加えるような攻撃や、自動逆難読化ツールについての耐性の検討を更に行うことが必要であり、また、基本アイディアを元にしたさらなる手法の提案を行うことも必要である

参考文献

[1] 門田暁人ら：“ 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 (1997.7).

[2] C.Collberg et al.：“ A Taxonomy of Obfuscating Transformations ”, TR148 , Dept. of Computer Science, University of Auckland (1997.7).

[3] Jien-Tsai Chan et al.：“ Advanced Obfuscation Techniques for Java Bytecode ”, Journal of Systems and Software, Vol.71, pp.1-10 (2004.4).

[4] J.M.Memon et al.：“ Preventing Reverse Engineering Threat in Java Using Byte Code Obfuscation Techniques ”, ICET’06, pp.689-694 (2006.11).

[5] S.Cimato et al.：“ Overcoming the Obfuscation of Java Programs by Identifier Renaming ”, Journal of Systems and Software, Vol.78, pp.60-72 (2005.10).