Java クラスファイル難読化ツール DonQuixote

DonQuixote: A tool for obfuscating Java class files

玉田 春昭[*] **中村 匡秀**[†] **門田 暁人**[‡] **松本 健一**[§]

あらまし 近年，難読化が注目されており，様々な難読化手法が提案されている．しかし，既存の難読化ツールは年々発表される数多くの難読化手法に対応できるような拡張性を持つものは存在しない．そこで我々は，様々な難読化手法が導入可能であるように，柔軟に拡張可能な難読化ツールの要件をまとめ，難読化ツール DonQuixote を開発した．DonQuixote はあらゆる難読化手法に対応可能であり，複数の難読化手法を任意の順序で組み合わせることが可能である．

1 はじめに

近年，難読化の需要が高まってきており，様々な難読化ツールが開発，提供されている [1,2] ツールと同様に多くの難読化手法も次々と提案されている [3–5]．これらの難読化手法は，プログラム中に含まれるある特定の情報を隠蔽する目的があり，それぞれの手法で隠すことのできる情報は異なっている．

隠したい情報により適用すべき難読化手法は異なるべきであるが，既存の難読化ツールでは，いくつかの難読化手法が一定の決まったステップに従って実行される．そのため，ユーザが自由に難読化手法の適用順を変更することや，特定の一つだけの難読化手法を適用すること，そして，新たな難読化手法を追加することは非常に困難となっている．

一方，ソフトウェア保護技術の研究・学習のためのツールに Sandmark がある [6]．Sandmark は，新たな難読化手法を追加することが可能である点が評価できる．しかし，複数の難読化手法を一括して組み合わせることができない点が問題である．加えて，Sandmark が入力として受け入れるのは jar ファイルのみであり，jar ファイルの一部のみを難読化したい場合や，jar ファイル内のクラスファイルごとに異なる難読化手法を適用することが不可能である．

このように，現在流通している難読化ツールは実用面の観点から見ると足りない部分が存在する．そこで，我々は上記の問題点を整理し，また，難読化ツールに求められている機能をまとめ，既存の難読化ツールの問題点を解決する Java 言語用の難読化ツール，DonQuixote を開発した [7]．DonQuixote はあらゆる難読化手法を組み込めるよう拡張性に富んでおり，また，難読化手法の組み合わせも柔軟に変更できる柔軟性も併せ持つ．

2 難読化手法

難読化とは，あるプログラムを理解が困難な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解析から保護するものである．難読化にはプログラムに含まれる何を隠すのかによって，様々な手法が存在する．ここでは DonQuixote 用に実装した難読化手法について述べる．それぞれの難読化の例を図 1 に示す．

- **名前変換 (Symbol Name Renaming)**
  与えられたクラスに含まれるシンボル名 (クラス名，フィールド名，メソッド

[*]Haruaki Tamada, 奈良先端科学技術大学院大学

[†]Masahide Nakamura, 奈良先端科学技術大学院大学

[‡]Akito Monden, 奈良先端科学技術大学院大学

[§]Ken-ichi Matsumoto, 奈良先端科学技術大学院大学

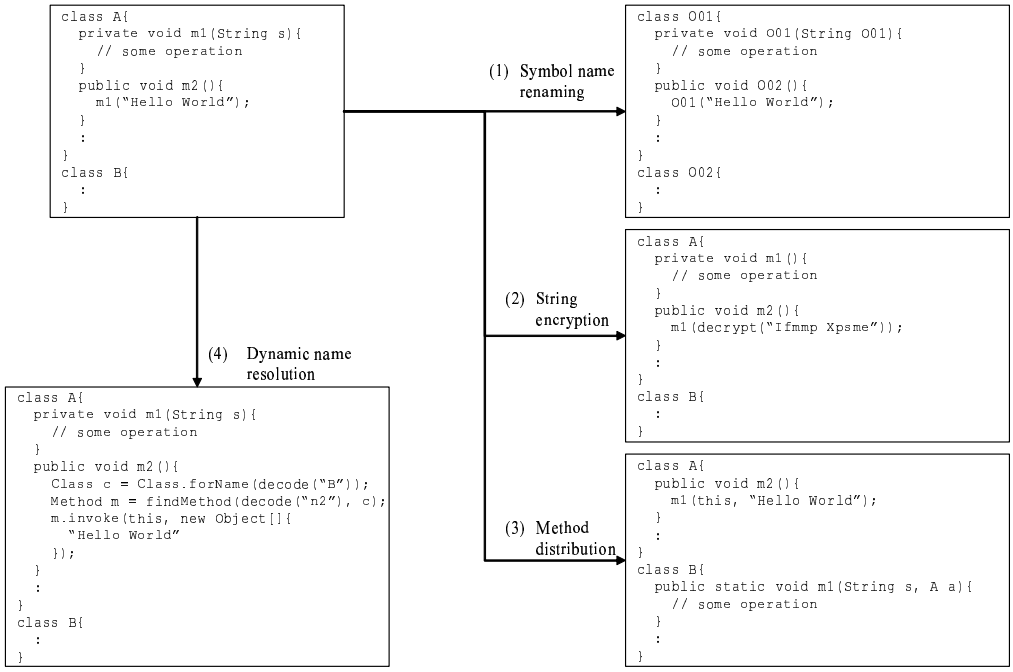

図 1　各難読化の例

名) の定義を変更し，意味のない名前にする難読化手法である．名前が定義されている個所を変更するため，システムが提供する API に含まれる名前を変更することはできず，ユーザが定義した名前のみが変換の対象となる．例を図 1 の (1) に示す．例では名前を O と 2 桁の数値の組み合わせに変換している．

- **文字列暗号化 (String Encryption)**
  与えられたクラス内にある文字列リテラルを予め暗号化する難読化の 1 つである．実行時，文字列が参照されるときに毎回復号処理が行われる．図 1 の (2) に例を示す．例では文字列を鍵 1 のシーザー暗号で暗号化している．そして，実行時に複合メソッド decrypt により暗号化された文字列が複合される．
- **メソッド分散 (Method Distribution)**
  与えられたクラスの集合からランダムに選択したメソッドを別のクラスに移動させる難読化手法である [5]．図 1 の (3) に例を示す．メソッド移動後も正常に処理が行われるように，呼び出されるメソッドのシグネチャを public static に変更し，メソッドを呼び出す側も移動先のクラスの静的メソッドを呼び出すように変更する．そして，移動するメソッドの内部からメソッド移動元のクラスのフィールドが参照されても良いように，引数にメソッド移動元のオブジェクトを渡すように変更する．
- **動的名前解決 (Dynamic Name Resolution)**
  メソッド呼び出しなどの名前解決を全て動的に行うように変更し，名前が使われている部分を難読化する手法 [8]．まず，メソッドの呼び出しやフィールドの参照，代入部を動的に行うように変更する．そして，動的解決するために与えられる文字列を暗号化し，実行時に復号するように変更することで，呼び出すメソッドを隠蔽する難読化手法である．例を図 1 の (4) に示す．例では名前を鍵 1 のシーザー暗号で暗号化し，クラスやメソッドをリフレクションを用いて取得し，メソッドを実行している．

3 DonQuixote

Java 言語を対象とした既存の難読化ツールの問題点を解決するため，我々は新たな難読化ツールとして DonQuixote を開発した．その詳細を以下に述べる．前提として，対象は Java クラスファイルとする．

3.1 要件

既存の難読化ツールの問題点として，(a) 新たな難読化手法が導入されるような，第 3 者による拡張が考慮されていない点，(b) 難読化の手順が決まっており，ユーザが自由に難読化手法を組み合わせることが不可能である点がある．これらを解決するため，DonQuixote に求められる要件を以下に示す．

要件 1 新たな難読化手法が容易に導入できること．

要件 2 複数の難読化手法を任意の順序で組み合わせることができること．

現在，多くの難読化手法が提案されているが，それら全ての手法を一人の開発者が作成することは現実的ではない．要件 1 は，これを解決するため，新たな難読化手法を容易に導入できるようにするものである．

また，要件 2 は難読化を決まった順序で行うのではなく，ユーザが自由に適用順序を入れ替えたり，組み合わせを変更したりを可能にするためのものである．難読化を適用する理由はクラッカーの攻撃からソフトウェアを守るためであり，守りたい情報により，効果的な難読化手法は一般的に異なる [9]．そのため，固定された一つの難読化手順ではなく，ユーザが自由に適用する難読化手法を決定できる必要がある．

上に挙げた機能要求に加えて，使いやすさの観点から，ツールの入力に関する要求も存在する．Java で書かれたプロダクトは多くの場面で，クラスファイルの集合をである jar ファイルの形式で扱われる．そのため，難読化ツールの入力に jar ファイル形式を受け入れることが必要であり，更に jar ファイル内の処理対象となるクラスファイルを絞り込む機能が必要である．

優先度の低い要件として，多くのインターフェースを持たせることも挙げられる．単純にツールとしての GUI や CUI のインターフェースを持たせるだけではなく，Eclipse [10] などの IDE や Apache Ant [11] などのビルドツールからも扱えるようなインターフェースを持たせることも望まれる．

3.2 アーキテクチャ

3.1 で述べたことを踏まえ，難読化ツール DonQuixote のアーキテクチャについて述べる．要件 1 と要件 2 を満たすために，難読化手法ごとにモジュールを作成することが考えられる．難読化手法をモジュールとして独立させることで，難読化手法をプラグインのように扱うことが可能となり，新たな難読化手法の導入も容易になる．そして，モジュールへの入力と出力をともに同じ形式のクラスファイルの集合とすることで，モジュールを任意の順序で実行させることも可能になる．そこで，**難読化モジュール (Obfuscation Module)** を導入する．この難読化モジュールは与えられたクラスファイルに処理を施し，その結果となるクラスファイルを出力するものである．多くの難読化手法は 1 クラスに対して処理を行うものであるため，難読化モジュールの入出力は共に 1 つのクラスファイルとする．ただし，メソッド分散のように，複数のクラスファイル間にまたがる難読化手法にも対応できるようにする必要がある．

DonQuixote のアーキテクチャを図 2 に示す．まず，DonQuixote は起動後，(1)

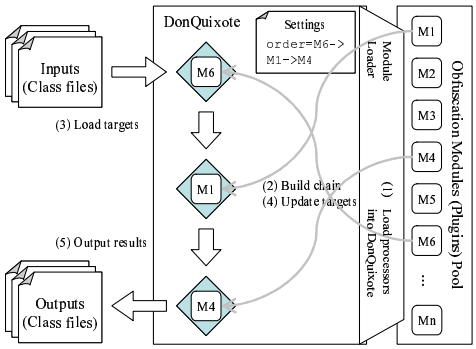

図 2　DonQuixote のアーキテクチャ

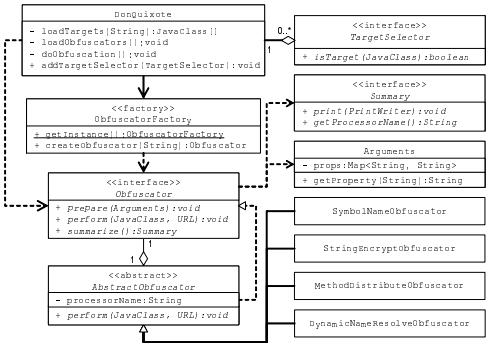

図 3　DonQuixote のクラス図

所定の場所にある難読化モジュールの集合から適用される難読化モジュールをロードし，(2) 難読化モジュールのチェインを作成する．その後，(3) 難読化対象のクラスファイルをロードし，(4) 順に各難読化モジュールへロードしたクラスファイルを渡し，クラスファイルを変換していく．最後に (5) 変換後のクラスファイルをファイルシステムへ出力し，DonQuixote を終了する．

3.3　設計

DonQuixote のクラス図を図 3 に示す．`Obfuscator` インターフェースが一つの難読化モジュールを表し，その具象クラスに 2 で述べた各難読化を行うクラスが存在する．難読化モジュールのチェインを作成するため，難読化モジュールの抽象クラス `AbstractObfuscator` にはデコレータパターンを採用した．また，難読化モジュールはそれぞれ名前を持っており，名前を指定することで `ObfuscatorFactory` により対応するモジュールがロードされる．

難読化は常に画一的に行うものではなく，モジュールにパラメータを与えて処理に変化を与えたい場合がある．例えば，文字列暗号の暗号化キーを自由に与えられるようにする必要がある．加えて，出力として得られるクラスファイルだけでは情報として不十分な場合がある．例えば，名前変換難読化の名前の変換前後のマッピングを得たいときである．そのため，パラメータの集合である `Arguments` をモジュールの初期化時に与え，モジュールでの処理終了後に `Summary` インターフェースを返すようにした．

3.4　実装

我々は実際に今までに述べた設計に基づき DonQuixote を実装し，今までに述べた通り 4 つの処理器を実装した．動作環境は Java SE 5 とし，Java クラスファイル操作ライブラリに BCEL 5.2 [12] を採用した．

4　ケーススタディ

4.1　各処理器の処理速度

DonQuixote の評価のため，実装した各処理器の実行時間を測定する．入力は jakarta-commons-digester 1.7 [13] の commons-digester.jar に含まれる 98 のクラスとした．実行環境は Windows XP Professional SP2，Intel Pentium M 1.4 GHz, 760 M RAM である．計測は同じ処理を 10 回行い，その平均とした．

実行に要した時間と，各クラスごとの平均時間を測定した結果を表 1 に示す．そ

表 1　処理に要した時間

	Process Time
Load class file	263.1 msec
Symbol Name Renaming	634.9 msec
String Encryption	1705.5 msec
Method Distribution	824.2 msec
Dynamic Name Resolution	2245.2 msec

れぞれ時間はミリ秒で表している．Load class files は入力となる 98 のクラスがロードされるまでの時間，その他は各処理器が初期化される直前から，結果となる Summary オブジェクトを返すまでの時間である．

表からわかるように，処理器により実行速度にばらつきがあるものの，処理に要する時間は String Encryption，Dynamic Name Resolution を除いて 1 秒以下となっている．

4.2　難読化例

ここでは，複数の難読化手法を組み合わせて実際に難読化を行う．図 4 に示すフィボナッチ数列の n 項目を計算する Java プログラムを名前変換，動的名前解決，文字列暗号化の順序で難読化した結果を図 5 に示す．ただし，例ではソースコードを示しているが，実際はクラスファイルを直接変更しているため，実際の処理とは異なる部分がある．また，文字列暗号化と，動的名前解決に DES を用いている．スペースの都合上暗号化後の文字列は一部省略している．

図 5 では，まず名前難読化を適用し，名前の定義部を難読化したため，図 4 のクラス宣言のクラス名，メソッド名，メソッド引数の変数名が変更されている．そして，動的名前解決難読化により，全てのメソッド呼び出しが DynamicCaller 経由で行われるように変更され，メソッド名が隠される．最後に文字列暗号化が行われるため，動的名前解決難読化により暗号化されたメソッド名などの名前が暗号化され，

```
public class Fibonacci{
  public int fibonacci(int n){
    if(n < 1) throw new IllegalArgumentException();
    if(n == 1 || n == 2) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
  }
}
```

図 4　難読化前のソースコード

```
public class O0001{
  public int O0002(int O0003){
    if(O0003 < 1)
      throw (Throwable)DynamicCaller.newInstance(
        new Object[0],
        __decrypt("cf00ae0044....fef1ea21b8")
      );
    if(O0003 == 1 || O0003 == 2) return 1;
    return
      ((Integer)DynamicCaller.invoke(this,
        new Object[] { new Integer(O0003 - 1), },
        __decrypt("69ad693eab...fef1ea21b8"),
        __decrypt("b81e592ed3...fef1ea21b8"))).intValue()+
      ((Integer)DynamicCaller.invoke(this,
        new Object[] { new Integer(O0003 - 2), },
        __decrypt("69ad693eab...fef1ea21b8"),
        __decrypt("b81e592ed3...fef1ea21b8"))).intValue();
  }
  private static String __decrypt(String string){
    // decrypt given strings
    return new String(cipher.doFinal(...));
  }
}
```

図 5　図 4 を名前変換，動的名前解決，文字列暗号化の順で難読化した例

```
public class O0001{
  public int O0003(int O0002){
    if(O0002 < 1)
      throw (Throwable)O0009.O0018(
        new Object[0], "92da227987....beec15702e"
      );
    if(O0002 == 1 || O0002 == 2) return 1;
    return
      ((Integer)O0009.O0021(this,
        new Object[] { new Integer(O0002 - 1), },
        "1774e4dbaa....5c4f1706f4"
        "80a1debba9....5c4f1706f4")).intValue() +
      ((Integer)O0009.O0021(this,
        new Object[] { new Integer(O0002 - 2), },
        "1774e4dbaa....5c4f1706f4"
        "80a1debba9....5c4f1706f4")).intValue();
  }
}
```

図 6　図 4 を文字列暗号化，動的名前解決，名前変換の順で難読化した例

実行時に復号されるように処理が追加されている．このように複数の難読化手法を自由に組み合わせて適用することが可能である．

また，難読化の適用順序を変更し，文字列暗号化，動的名前解決，名前変換の順に同じく図 4 のプログラムを難読化した．結果を図 6 に示す．難読化の適用順序を変更したため，図 5 とは異なる結果となっている．しかし，難読化の適用順序に問題があるため図 6 に示したクラスファイルは実行することができない．

最初に文字列暗号化を行うが，対象となる文字列が存在しないため，何も処理が行われない．そして，動的名前解決により全てのメソッド呼び出しが DynamicCaller 経由になり，呼び出すメソッドが文字列となる．その後，名前変換を行い，メソッド名やクラス名の定義を変更される．しかし，動的名前解決では変換前のメソッド名で名前解決を行おうとため，実行時にメソッド定義が見つからないエラーによりアプリケーションが異常終了する．

このように，任意に難読化手法を組み合わせることが可能であるが，適用順序により，効果のある組み合わせや，意味のない組み合わせ，そして，不具合がおきる組み合わせが存在する．組み合わせの効果については今度調査する必要がある．

5 まとめ

本稿では，近年の難読化ツールの問題点を述べ，その問題点を解決する上で求められる要件をまとめた．また，実践的なツールとして DonQuixote と，それに組み込む 4 つの難読化モジュールを実装した．それらの難読化モジュールは適用する順序をそれぞれ自由に組み合わせることが可能である．そして，DonQuixote を評価するため，実行速度を計測した結果，実用的レベルの速度で処理できることを示した．

今後の課題として，GUI などユーザインターフェース部分の充実や，難読化モジュールの拡充などが挙げられる．

参考文献

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

[2] Eric Lafortune. Proguard, May 2006. http://proguard.sourceforge.net/.

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

[4] 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.

[5] 福島和英, 田端利宏, 田中俊昭, 櫻井幸一. クラス構造変換手法を用いた Java プログラムへの利用者識別情報の埋め込み手法. 情報処理学会論文誌 多様な社会的責任を担うコンピュータセキュリティ技術特集号, Vol. 46, No. 8, pp. 2042–2052, August 2005.

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

[7] Haruaki Tamada. DonQuixote: A framework for obfuscating/translating java byte code, 2006. http://donquixote.cafebabe.jp/.

[8] 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.

[9] Hiroki Yamauchi, Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken ichi Matsumoto. Software obfuscation from crackers ’ viewpoint. In *Proc. IASTED International Conference on Advances in Computer Science and Technology (IASTED ACST 2006)*, pp. 286–291, January 2006. Puerto Vallarta, Mexico.

[10] The Eclipse Foundation. Eclipse, 2006. http://www.eclipse.org/.

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

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

[13] Apache Software Foundation. Jakarta commons digester, June 2005. http://jakarta.apache.org/commons/digester/.