CB-002

ランダムフォレストを用いた難読化されたコードのステルス評価の検討

A Study on the Evaluation of the Stealth of Obfuscated Code Based on Random Forest

北岡 哲哉 * Tetsuya Kitaoka　神崎 雄一郎 * Yuichiro Kanzaki　森川 みどり † Midori Morikawa　門田 暁人 ‡ Akito Monden

1. はじめに

ソフトウェアに対する Man-At-The-End 攻撃，すなわち，ソフトウェアの実行可能コードを所有するエンドユーザによるコードの不正な解析・改ざん行為 [1] は，ユーザのローカル環境で実行されるソフトウェアへの脅威となっている．Man-At-The-End 攻撃からソフトウェアを保護する一手段として，コードの難読化，すなわち，コードの意味を保ったまま解析が困難な (攻撃者の解析に多くの時間を要する) コードに変形する技術が知られており，演算表現の複雑化や制御構造の平滑化など，従来様々な難読化方法が提案されている [2, 3]．また，近年では，Tigress [4, 5] や Obfuscator-LLVM [6] といった無償の難読化ツールも複数公開されている．

難読化されたコードの有効性を評価する指標として，Collberg らは，理解の困難さ (potency)，自動逆難読化の困難さ (resilience)，計算量のオーバーヘッドの低さ (cost) と並んで，ステルスの高さ (stealth) を挙げている [7]．難読化されたコードのステルスとは，難読化されていないコードとの区別のつきにくさ，すなわち，攻撃者による難読化されたコードの発見の困難さを意味する [2, 7]．攻撃者に知られたくない重要なコードやデータを含むプログラムの一部を難読化によって保護している場合には，攻撃者に攻撃の糸口を容易に与えないために，難読化変形されたコードのステルスを高く保つことは重要であり，そのためには，難読化されたコードのステルスを評価する方法が求められる．

そこで本研究では，難読化されたコードのステルスを機械学習を用いて評価する方法を提案する．具体的には，アセンブリコードが難読化によって変形されているかどうかを判定 (2 クラス分類) するモデルをランダムフォレストによって構築し，難読化された任意のアセンブリコードのステルスの評価に用いる．ケーススタディでは，演算表現の複雑化や制御構造の平滑化などの既存の難読化方法を対象にした実験を通して，提案方法の有効性や難読化されたコードの特徴について議論する．

* 熊本高等専門学校，National Institute of Technology, Kumamoto College
† 株式会社ワイズ・リーディング， Y's Reading Inc.
‡ 岡山大学，Okayama University

2. 関連研究

近年，難読化されたコードの性質を，機械学習を用いて評価する方法がいくつか提案されている．例えば，Banescu らは，難読化されたコードに対するシンボリック実行攻撃への耐性評価を行うための，機械学習を用いたフレームワークを提案している [8]．また，磯部らは，名前難読化が適用されたコードの逆難読化に対する耐性を，ランダムフォレストを用いて評価する方法を提案している [9]．本研究では，これらのテーマと異なり，難読化されたコードのステルスの評価方法を提案する．

難読化されたコードのステルス評価に関する先行研究として，神崎らは，N-gram モデルによって得られるコードの「不自然さ」にもとづいてコードのステルスを評価する方法を提案している [10]．一方，本研究では，コードを構成する各命令の出現頻度を用いて構築した，難読化されているかどうかを判定する機械学習のモデルによって，コードのステルス評価を試みる．先行研究では，まれにしか出現しないような不自然な命令系列が出現した場合にステルスが低いと評価されるのに対し，提案方法では，命令としてはありふれたものであっても，その出現頻度が不自然に多いもしくは少ない場合にステルスが低いと評価されることとなる．

3. 提案方法

本研究で提案するコードのステルス評価方法について，基本アイデアを述べる．本稿では，一連のアセンブリ命令 (例えば，関数や基本ブロック) をコードと呼ぶ．まず，コードが難読化されているかどうかを判定する分類モデルを，コードの命令構成 (コードに含まれる各命令の出現頻度) に着目してランダムフォレストによって構築する．ある難読化されたコードが，構築したモデルによって「難読化されている」と分類された場合は，そのコードの難読化による変形が目立つ (難読化されていないものと区別がつきやすい変形が生じている) とみなして，そのコードのステルスが低い (unstealthy) と考える．同様に，難読化されたコードがモデルに「難読化されていない」と分類された場合は，ステルスが高い (stealthy) と考える．

提案方法による具体的なステルス評価の流れを図 1 に示す．まず，難読化されていないコードと難読化された

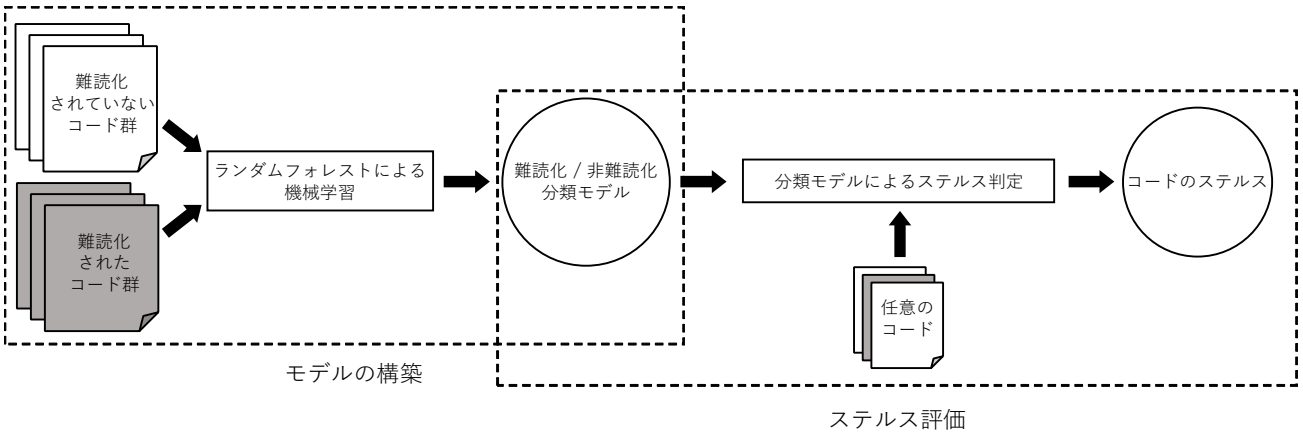

図 1　提案方法によるステルス評価の流れ

コードを多数用意し，ランダムフォレストによる機械学習を行って，コードが難読化されているかどうかの 2 クラスに分類するモデルを構築する．訓練データとして与える特徴は，ここではコードを構成する各命令の出現頻度 (コードの命令数で正規化したもの) とする．評価対象となるコードが，構築した分類モデルによって難読化されていると分類されるかどうかで，そのコードのステルスを評価する．

4. ケーススタディ

4.1 概要

ここでは，3 章で述べた提案方法の流れ (図 1) に沿って，モデルの構築とステルスの評価に関する実験を行う．本ケーススタディの目的は，(1) 各命令の出現頻度を説明変数としたランダムフォレストの分類モデルによってコードが難読化されているかどうかをどの程度の精度で判定できるか，(2) コードがどのような命令構成を持てばステルスが低いまたは高いと判定されるか，といった点を実験結果を通して議論することである．

4.2 コードの準備

まず，モデルの構築に用いたコード群について説明する．本ケーススタディでは，文献 [11] に掲載されている 254 個の C 言語プログラムを用いた．これらは，データのソートや探索，素因数分解など，様々なアルゴリズムが実装されたものである．これらの各プログラムを関数単位で分割して得られる 1,272 個のコードを，難読化されていないコード群として用いた．また，これらのコード群を，既存の難読化方法によってそれぞれ変形したものを，難読化されたコード群とした．適用した難読化方法は，静的難読化方法としてよく知られている，演算表現の複雑化 [4]，制御構造の平滑化 [12]，opaque predicate [2] の挿入の 3 つであり，これらの難読化方法が実装された難読化ツール Tigress [4, 5] を用いて，関数単位で難読化した．一部，Tigress による難読化が失敗したものがあったが，難読化前の 1,272 個のコードに対して，3,030 個の難読化されたコード群が得られた．各コードは GCC [*1]によって一度コンパイルした後，バイナリ解析ツールの IDA [*2]を用いて逆アセンブルすることでアセンブリレベルでのコード群を用意した．

4.3 モデルの構築

次に，コードに含まれる各命令の出現頻度を説明変数として，コードが難読化されているかどうかを分類する

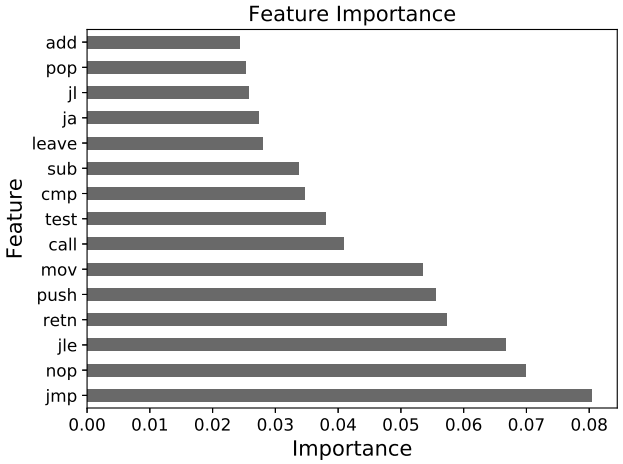

図 2　構築したモデルの特徴の重要度 (上位 15 個)

[*1] GCC, the GNU Compiler Collection: `https://gcc.gnu.org`

[*2] Hex-Rays IDA: `https://www.hex-rays.com/products/ida/`

表 1　構築したモデルの分類性能

検証方法	評価項目	値
ホールドアウト	Accuracy	0.960
	F1 score	0.972
層化 10 分割交差	Accuracy	0.951
	F1 score	0.941

モデルを構築する．具体的には，4.2 で得られた両コード群の各コードについて，各アセンブリ命令のオペコードの出現頻度 (コード内の命令数で正規化したもの) と，難読化されたかどうかのラベルを含むデータセットを準備し，それを用いてランダムフォレストによる学習を行うことで，モデルを構築する．なお，ここでは，データセットを訓練用とテスト用で 7:3 に分割して学習を行った．

なお，モデルの構築やモデルの検証 (後述) には，Python の機械学習ライブラリ scikit-learn [*3]を用いた．同ライブラリによって得られた構築したモデルの特徴の重要度 (上位 15 個) を図 2 に示す．図 2 から，今回構築したモデルは `jmp` 命令，`nop` 命令，`jle` 命令などの出現頻度が，難読化されているかどうかを判別する際に重要とみなされていることがわかる．

4.4 モデルの検証

構築したモデルの精度の検証について述べる．まず，4.3 で述べた，テスト用に分けられたデータ (データセット全体の 3 割分) を用いて，正解率 (accuracy) と F1 値 (F1 score) を測定した (ホールドアウト)．さらに，分割前のデータセット全体に対して，層化 10 分割交差検証にて同様に正解率と F1 値を測定した．結果を表 1 に示す．表 1 から，どちらの検証方法でも，モデルの正解率は 95% 以上，F1 値は 94% 以上であった．

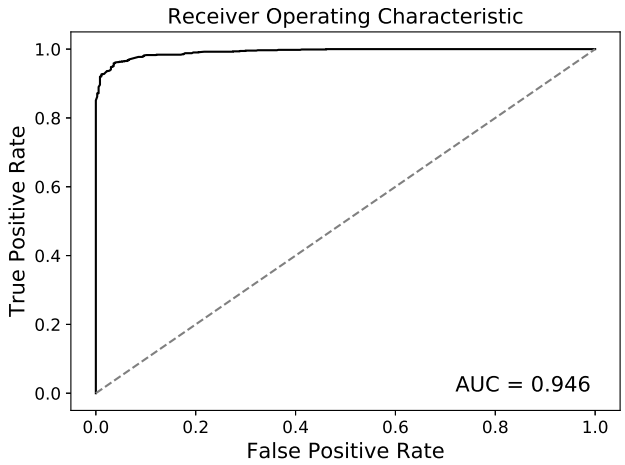

図 3　ROC 曲線

加えて，構築したモデルの ROC 曲線 [13] を図 3 に示す．図 3 にある True Positive Rate は，難読化されたコードをモデルが「難読化されている」と正しく分類した確率であり，False Positive Rate は，難読化されていないコードをモデルが「難読化されている」と誤って分類した確率である．ROC 曲線の AUC (Area Under the Curve) は 0.946 と高く，構築した分類モデルの性能が良いことを示している．

以上より，今回の実験の範囲内では，各命令の出現頻度を特徴としたランダムフォレストの分類モデルによって，難読化されたコードと難読化されていないコードを区別できる場合が多いといえる．

```
push
mov
call
test
js
pxor
cvtsi2d
jmp
mov
shr
and
or
pxor
cvtsi2d
addsd
movsd
mulsd
pop
retn
```

(a) 難読化前

```
push    mov
mov     shr
sub     and
mov     or
mov     pxor
test    cvtsi2sd
jz      addsd
cmp     movsd
jnz     mulsd
call    jmp
mov     jmp
mov     leave
jmp     retn
mov
test
js
pxor
cvtsi2sd
jmp
```

(b) 難読化後

図 4　難読化前後のコード (オペコード列) の例

4.5 議論

本ケーススタディにおいては，テストデータに含まれる難読化されたコードのほとんどが，分類モデルに「難読化されている」と判定され，ステルスが低いと評価された．このようなコードの例として，M 系列乱数を生成するプログラム中の 1 つの関数のコードと，それを制御構造の平滑化によって難読化したコードの各オペコード列を図 4 に示す．図 4 から，難読化されることで `mov` 命令，`jmp` 命令などが増加していることがわかる．これは制御構造の平滑化が，分岐数の多い条件分岐 (C 言語での `switch` 文) を積極的に用いることに起因しているが，これらの命令の出現頻度はいずれも図 2 において重要度

[*3] scikit-learn: `https://scikit-learn.org/`

が高い特徴とみなされている．この例からも，ランダムフォレストによって，難読化の前後で出現頻度が大きく変化する命令が正しく特定されていることがわかる．

一方，難読化されているにもかかわらずステルスが高いとみなされたものは 21 個存在し，それらはすべて演算表現の複雑化によって難読化されたコードであった．これらのソースコードを確認すると，難読化前と比べて，変数のキャストなどの細かな表現の変形は行われているものの，演算表現は複雑になっていなかった．したがって，これらは演算表現の複雑化による難読化変形が十分に機能していないため，難読化されていないとみなされたと考えられる．

以上のように，今回の実験の範囲内では，難読化されたコードのほとんどが，機械学習によって難読化されていないコードと判定される，すなわち，ステルスが低いと判定される結果になった．このような分類モデルでも難読化されていないと判定されるような難読化の変形方法や，難読化後にステルスを向上させるコードの変形方法を提案することは興味深いと考える．

5. おわりに

本稿では，ランダムフォレストによる機械学習で構築した 2 クラス分類モデルによって，難読化されたコードのステルスを評価する方法を提案した．ケーススタディでは，既存のプログラムや難読化方法を用いてモデルを構築し，コードのステルスを評価する実験を行った．結果として，難読化されたコードの多くが難読化されている (ステルスが低い) と判定された．

今後の課題として，訓練用のデータの数や適用する難読化方法の種類を増やした場合や，アセンブリ命令のオペランドを考慮した場合についてのケーススタディを行うことがあげられる．また，ランダムフォレスト以外の機械学習のアルゴリズムを用いて同様の実験を行い，結果を比較することも検討している．

謝辞

本研究は，JSPS 科研費 JP19K11916 の助成を受けたものである．

参考文献

[1] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski, “Software protection (guest editors’ introduction),” IEEE Software, Special Issue on Software Protection (March/April 2011), vol.28, no.2, pp.24–27, March 2011.

[2] C. Collberg and J. Nagra, Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Program Protection, Addison-Wesley Professional, 2009.

[3] 門田暁人，C. Thomborson，“ソフトウェアプロテクションの技術動向 (前編) - ソフトウェア単体での耐タンパー化技術，” 情報処理，vol.46，no.4，pp.431–437，April 2005.

[4] C. Collberg, “The Tigress C diversifier/obfuscator,” http://tigress.cs.arizona.edu/ (accessed: June 2019).

[5] C. Collberg, S. Martin, J. Myers, and J. Nagra, “Distributed application tamper detection via continuous software updates,” Proceedings of the 28th Annual Computer Security Applications Conference, pp.319–328, ACSAC ’12, 2012.

[6] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, “Obfuscator-LLVM – software protection for the masses,” Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO’15), pp.3–9, 2015.

[7] C. Collberg, C. Thomborson, and D. Low, “Manufacturing cheap, resilient, and stealthy opaque constructs,” ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL98), pp.184–196, Jan. 1998.

[8] S. Banescu, C. Collberg, and A. Pretschner, “Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning,” 26th USENIX Security Symposium (USENIX Security 17), pp.661–678, 2017.

[9] 磯部陽介，玉田春昭，“ランダムフォレストを用いた名前難読化の耐タンパ化性能の評価，” 情報処理学会論文誌，vol.60，no.4，pp.1063–1074，April 2019.

[10] 神崎雄一郎，尾上栄浩，門田暁人，“コードの「不自然さ」に基づくソフトウェア保護機構のステルシネス評価，” 情報処理学会論文誌，vol.55，no.2，pp.1005–1015，Feb. 2014.

[11] 奥村晴彦，C 言語による標準アルゴリズム事典，技術評論社，2018.

[12] C. Wang, J. Hill, J.C. Knight, and J.W. Davidson, “Protection of software-based survivability mechanisms,” Proceedings of the 2001 International Conference on Dependable Systems and Networks, pp.193–202, 2001.

[13] T. Fawcett, “An introduction to ROC analysis,” Pattern Recognition Letters, vol.27, no.8, pp.861–874, 2006.