コード難読化ツールの信頼性を評価するフレームワークの検討

Towards a Framework to Evaluate the Reliability of Code Obfuscation Tools

北岡 哲哉 * 神崎 雄一郎 † 石尾 隆 ‡ 嶋利 一真 § 松本 健一 ¶

あらまし ソフトウェアを解析や改ざんから保護するためのツールとして，コード難読化を実装したツール（難読化ツール）が多数提案されている．しかし，難読化ツールが対象コードに不具合を生じさせずに解析を困難にできるか，すなわち「信頼できる」ものかを調べる方法は明らかになっていない．本研究では，難読化ツールの信頼性を評価するための方法を検討する．具体的には，信頼性の高い難読化ツールかを判断するためのメトリクスとして，与えられたコードの機能を維持したまま難読化を適用できる度合いを表す機能維持性と，与えられたコードの内容を変化させる度合いを表すコード変形性という 2 つの指標を提案し，多数のプログラムを用いてこれらを実証的に評価するためのフレームワークを提案する．提案したフレームワークを用いた実験では，実際に公開されている 2 種類の難読化ツールを対象に機能維持性とコード変形性の評価を行い，得られた結果から，実験対象の難読化ツールの信頼性について議論する．

1 はじめに

ソフトウェアに対する不正な解析および改ざん攻撃 [1] の成功を遅らせる方法として，コード難読化 (code obfuscation) が広く用いられている．コード難読化は，与えられたプログラムコードを，コードの機能を維持したまま人間や機械による解析が困難な形に変換する技術である．コード難読化を実装したツール（以下，難読化ツールと呼ぶ）として，多数の商用のツールに加え，Tigress [2,3] や Obfuscator-LLVM [4] といった無償のツールも活発に開発され，公開されている．

コード難読化は，変換前のコードが持つ入力と出力の対応関係（入出力関係）が，変換後のコードでも維持されることが前提となる．一方で，コード難読化では，制御構造の平滑化，コードの仮想化，実行時コード生成など，プログラムの構造を大きく変形し得る様々な変換を行う [5]．そのため，難読化ツールを適用することで，ソフトウェア開発者が意図していない不具合がコードに生じる可能性は否定できない．実際の Android アプリケーションの開発者を対象にしたコード難読化に関する調査においても，コード難読化によって自身が作成したソフトウェアに不具合が生じる（ソフトウェアが「破壊」されてしまう）ことを懸念する声が報告されている [6]．難読化ツールによってソフトウェアが適切に保護されるためには，難読化ツールが不具合を生じさせずに解析を困難にできるか，すなわち「信頼できる」ものかを把握することが重要である．しかし，個々の難読化ツールの開発者が，難読化対象となる様々なソフトウェアを収集し，信頼性の評価を行うことは難しい．

そこで本研究では，難読化ツールの信頼性を評価するための方法を提案する．本稿では，その第 1 の基準として，機能維持性，すなわち，様々なプログラムコード

*Tetsuya Kitaoka, 奈良先端科学技術大学院大学
†Yuichiro Kanzaki, 熊本高等専門学校
‡Takashi Ishio, 奈良先端科学技術大学院大学
§Kazumasa Shimari, 奈良先端科学技術大学院大学
¶Kenichi Matsumoto, 奈良先端科学技術大学院大学

に対して，不具合を生じさせることなく元来の機能を維持してコードの変換を行える度合いを表す指標を設ける．機能維持性は，多数のコードをツールに与えたときに，難読化前後で同一の入出力関係を保っているコードの割合によって評価する．

難読化されたコードが元来の機能を維持できているとしても，元来のコードと内容が同一であったり，ごくわずかな変化しかない場合は，コードの解析が困難になったことを期待できるとはいい難い．そのため，第 2 の基準として，コード変形性，すなわち，様々なプログラムコードに対して，コードの内容を変化させた度合いを表す指標を設ける．コード変形性は，多数のコードをツールに与えたときに，各コードの難読化前後のコード間の距離（類似度の低さ）の平均によって評価する．

本研究では，このような機能維持性やコード変形性といった信頼性に関するメトリクスを定義し，多数のプログラムを用いてそれらを実証的に評価するためのフレームワークを提案する．また，実際に公開されている 2 種類の難読化ツールを対象に機能維持性やコード変形性を評価する実験を行い，得られた結果から，実験対象の難読化ツールの信頼性について議論する．

本稿では，2 章で関連する研究を紹介する．3 章で提案手法について述べ，4 章で実験について述べた後，5 章で実験結果について議論を行う．最後に 6 章でまとめと今後の課題について述べる．

2 関連研究

コード難読化の有効性，すなわち解析攻撃に対する強さを評価する研究は従来さかんに行われており，近年も増加傾向にある [7]．例えば，Banescu らは，シンボリック実行を用いた自動解析攻撃に対する防御方法として，既存の難読化方法の有効性を，難読化されたコードの実際の解析時間によって議論している [8]．また，難読化された Java プログラムを理解・改ざんする困難さを，実際に人間が解析する時間に基づいて評価する手法 [9] や，難読化機構が攻撃者に発見されにくい度合い（ステルス性）を確率的言語モデルを用いて評価する手法 [10]，コードに難読化を適用した難読化ツールの特定がどれだけ困難かを Java バイトコードのオペコード列の出現頻度の類似性をもとに評価する手法 [11] も提案されている．加えて，文献 [12] では，名前難読化の逆変換の困難さを評価する手法が提案されている．この研究では，Java バイトコードのオペコード列が難読化ツールの適用によってどの程度変化するのかを，難読化前後のオペコード列の編集距離によって算出する実験を行っており，この点においてコード変形性の評価にも関連がある．

本研究は，特定の攻撃モデルに対する防御の強さではなく，「元来の機能を維持したままコードの内容を十分に変化させることができるか」という難読化方法（難読化ツール）の基本的な品質を評価することを試みる．提案手法は，一定の攻撃モデルに特化した従来の有効性評価の方法を組み合わせることが可能であり，難読化方法やツールの性能を正確に評価するための一助になると考える．

3 提案手法

本研究では，難読化ツールの信頼性を評価することを目的として，難読化ツールの信頼性に関するメトリクスを定義し，これらを用いた評価フレームワークを提案する．難読化対象となるコード群は事前に与えられているものとして，難読化ツールを実行し，機能維持性とコード変形性という 2 つのメトリクスを計測する．難読化ツールは複数の難読化方法を実装している場合が一般的であるため，本フレームワークでは，難読化ツールが実装している難読化方法ごとに機能維持性とコード変形性を評価する．つまり，両メトリクスとも，難読化ツールが実装している 1 つの難読化方法に対して，1 つの値が出力される．

難読化方法 o を難読化対象コード群 P に対して適用した場合の機能維持性とコード変形性を以下のように定義する．

機能維持性 S： 難読化ツールが，与えられたコードの機能を維持したまま難読化を適用できる度合いを示す．このメトリクスは，難読化されたコードの入出力等価性，すなわち，入力値とコードに入力値を与えたときの出力が難読化前後ですべて一致しているようなコードの割合として，以下のように算出する．

$$S(o) = \frac{|\{p \in P | Equivalent(p, o(p))\}|}{|P|}$$

ここで，$o(p)$ は難読化方法 o を適用した結果である．また，$Equivalent(p, o(p))$ は，難読化前後のコードの組の入出力が同一であるときに真となる述語である．機能維持性は 1 となることが望ましい．難読化ツールが入出力等価性を満たすコードを生成できなかった場合，難読化後のコードは難読化前と同じ挙動をしないことを示すため，機能維持性は低下する．

コード変形性 D： 難読化ツールが，与えられたコードの内容を変化させる度合いを示す．このメトリクスは，難読化されたコードごとにコード間距離（コード類似度の低さ）を計測し，その平均から算出する．

$$D(o) = \frac{\sum_{p \in P} Distance(p,\ o(p))}{|P|}$$

難読化ツールが元来のコード内容や表現を十分に変化させていない場合，コード変形性が低くなる．

これらのメトリクスを算出するフレームワークの全体像を 図 1 に示す．本フレームワークへの入力は，難読化ツール，難読化対象のコード群とコード群に対応したシンボリック実行を行うスクリプトの 3 つで，出力は，機能維持性とコード変形性の 2 つである．提案するフレームワークは大きく 3 つのステップで構成される．Step Ⅰでは，まず，入力として与えられたコード群 P に対して，評価対象の難読化ツールの適用を行う．また，$Equivalent(p, o(p))$ の判定を行うための準備として，難読化前のコード群に対してシンボリック実行を適用し，コードの各出力に到達する入力値を探索し，得られた入出力関係をテストケースとして保存する．シンボリック実行による入力値の探索では，コードから発見した各実行パスごとに，実行パスが通過する具体的な入力値を 1 つずつ生成する．Step Ⅱでは，得られたテストケースを難読化後のコード群に対して実行し，それぞれのコードが入出力等価性を満たすかを検証し，機能維持性を算出する．

Step Ⅲでは，入力として与えられたコード群と，Step Ⅰで難読化されたコード群を用いて，コード変形性を算出する．コード間距離の計算には，様々なアプローチが考えられるが，本稿では，難読化前後での N-gram アセンブリ命令列からなる集合同士の Simpson 係数（Overlap coefficient） [13] をもとに測定する．Simpson 係数は，要素数が少ない方の集合の要素数に対する集合同士で共通する要素数の割合から求められるため，この値を 1 から減算することでコード間距離が測定可能である．また，Simpson 係数は一方の集合が別の集合の真部分集合であったときに算出値が 1 となる性質を持っているため，コード間距離が 0 となったときには難読化されたコードから元来のコードが予測されやすいと判断することができる．なお，Simpson 係数の測定において N-gram を採用した理由は，アセンブリ命令列の文脈を考慮して評価を行うためである．難読化前後の N-gram アセンブリ命令列の集合をそれぞれ $A(p)$，$A(o(p))$ としたとき，コード間距離は次の式で求められる．

$$Distance(p, o(p)) = 1 - \frac{|A(p) \cap A(o(p))|}{\min\left(|A(p)|, |A(o(p))|\right)}$$

コード間距離の測定のために，それぞれのコード群から難読化前後のコードを抽出し，実行可能コードをそれぞれ生成する．そして，生成された実行可能コードを逆アセンブルして得られた，N-gram アセンブリ命令列同士の Simpson 係数からコー

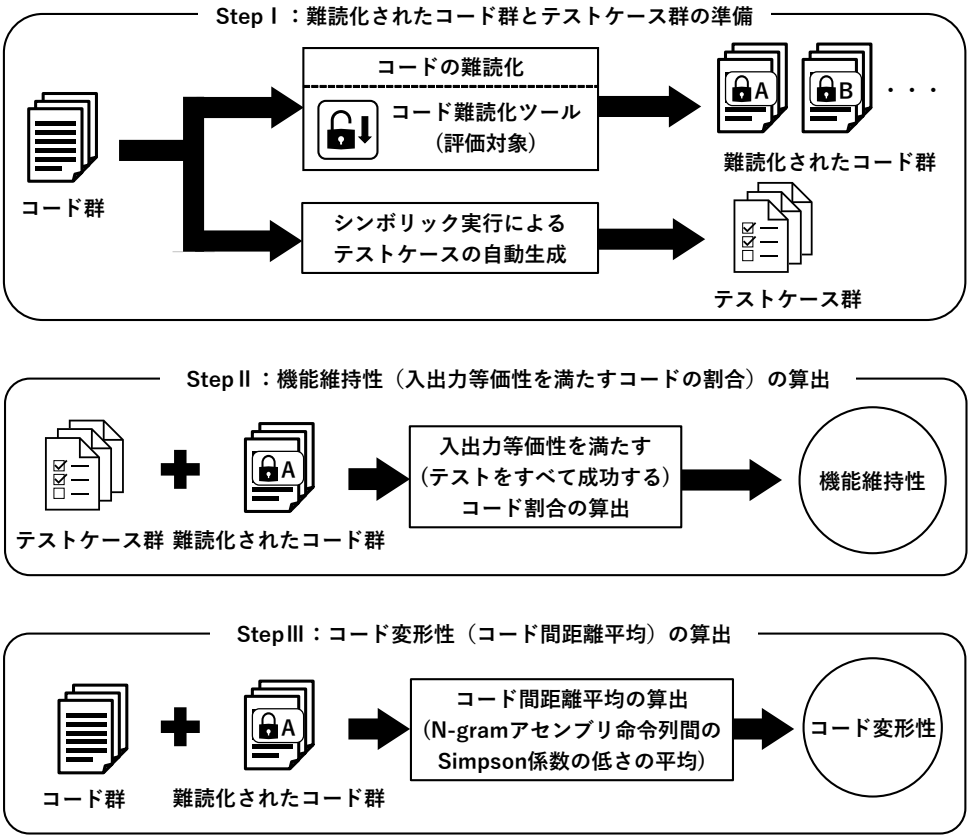

図 1: 提案するフレームワークの全体像

ド間距離を求める．その後，それぞれのコード群から測定したコード間距離の平均が，コード変形性として算出される．

以上の手順から，評価対象とする難読化ツールが実装している各難読化方法の信頼性に関する傾向を分析する．

4 実験

3 章で述べた提案手法のフレームワークを用いて，よく知られた難読化ツールである Tigress [2,3] や Obfuscator-LLVM [4] で実装されている難読化方法の機能維持性とコード変形性を評価する実験を行う．本実験で評価対象とする難読化方法や，難読化の適用対象となるコードの説明を次に示す．

4.1 実験方法

4.1.1 難読化ツールと難読化方法

評価対象の難読化ツールは，Tigress [2, 3] と Obfuscator-LLVM [4] の 2 つである．Tigress は C 言語のソースコードのレベルで難読化を行い，Obfuscator-LLVM は LLVM IR のレベルで難読化を行う．Tigress と Obfuscator-LLVM はどちらも複数の難読化方法を実装していることから，本実験では，対象とする難読化方法ごとに機能維持性とコード変形性を評価する．

今回の実験で対象とする Tigress の難読化方法を表 1(a) に，Obfuscator-LLVM

表 1: 実験対象の難読化ツールが実装している難読化方法

(a) Tigress が実装している難読化方法 (一部)

Abbr.	**Description**
EncA	Encode Arithmetic: 整数演算を複雑な表現に置換する
EncL	Encode Literals: 整数や文字列リテラルを不明瞭な表現に変換する
Flat	Flatten: 構造化された制御構造を平滑な形に変換する
AddO4	Add 4 Opaque predicates: Opaque predicate [5] を 4 つ挿入する
AddO16	Add 16 Opaque predicates: Opaque predicate [5] を 16 個挿入する
Virt	Virtualize: コードを仮想化する，すなわち独自のバイトコードとそのインタプリタの組合せに変換する

(b) Obfuscator-LLVM が実装している難読化方法

Abbr.	**Description**
BCF	Bogus Control Flow: Opaque predicate [5] を用いたダミーの条件分岐を追加する
CFF	Control Flow Flattening: 構造化された制御構造を平滑な形に変換する
ISub	Instructions Substitution: 加算・減算・論理演算などの演算を複雑な表現に変換する

の難読化方法を表 1(b) にそれぞれ示す．Tigress は，動的にコードを変形するものを含め多数の難読化方法の実装が公開されているが，今回は表 1(a) に挙げたものを用いる．Tigress の場合は，対象となる関数を指定して難読化を適用する一方で，Obfuscator-LLVM では，対象となる関数は指定せず，コード単位で難読化が適用される．

本実験では，表 1 に示した難読化方法を，1 つ単独で適用した場合と，2 つ組み合わせて適用した場合を評価する．本実験では，Banescu らによる，自動解析攻撃に対する難読化方法の有効性を評価する研究 [8] の実験で評価対象とされた組合せと同一のものを用いることとし，Tigress では計 28 種類，Obfuscator-LLVM では計 9 種類の難読化方法について，それぞれ機能維持性とコード変形性を評価する．難読化方法を組み合わせた場合の名称は，それらの方法を適用した順番に，難読化方法の名称を 2 つつなげて表記する．たとえば AddO16_EncA は，AddO16 を適用したあとに EncA を適用したことを意味する．また，同じ難読化方法を 2 回適用した場合は，x2 として表現している．

4.1.2 難読化対象のコードとテストケース

難読化の適用対象となるコード群は，Tigress に実装されている RandomFunc 機能によって自動生成された，公開されている 5,760 個の C 言語ソースコード [8] [1]から構築する．このコード群は，main 関数とランダムに生成された関数（RandomFunc 関数）を含んでおり，プログラムとしてはコマンドライン引数の値をもとに RandomFunc 関数が処理した値を出力するものとなっている．特定の値がプログラ

[1] `https://github.com/tum-i4/obfuscation-benchmarks/`

表 2: 難読化の適用対象となるコード群 (1,318 個) の情報

	Min	Max	Mean
論理 LOC	80	120	90.20
1 コードに対するテストケース数	3	5	3.107
1 コードに対するテストのカバレッジ (gcov の Line executed で計測)	90.00%	96.00%	92.49%

ムに与えられた場合，プログラムは “You win!” を出力する．

本実験では，シンボリック実行エンジンとして angr [14] を採用し，テストケースの自動生成を行う．具体的には，特定のコードの出力（“You win!” の文字列など）に到達する入力値をシンボリック実行によって探索させ，得られた入力値（コマンドライン引数）とそれに対する出力値の組で構成されるテストケースを生成する．ソースコードからコンパイルした実行ファイルにテストケースを与え，C 言語用のコードカバレッジの計測ツール gcov で求められた Lines executed の合計が 90%以上となるコードを，難読化の適用対象となるコード群として抽出した．結果として本実験では，1,318 個の C 言語のソースコードを難読化の適用対象となるコード群を得た．

難読化の適用対象となるコード群に関する情報を表 2 に示す．コード群全体で論理 LOC は 80 から 120 となっており，図 1 の Step. I によって，1 つのコードに対して 3〜5 つのテストケースが自動生成されている．図 1 の Step. II に該当する機能維持性の算出では，この生成したテストケースを実行し，元来のコードの機能に対応する実行が得られたかを計算する．

Tigress は特定の関数を選んで難読化を適用するツールであることから，main と RandomFunc のそれぞれに個別に難読化を適用し，部分的に難読化されたコードを 2 ファイル作成し，それらを個別に難読化したプログラムとみなして，機能維持性，コード変形性の計算に使用する（2,636 個のコード群から評価値を求める）．これらの実行可能コードは，GCC （バージョンは Red Hat 版の 8.5.0-12）を用いて生成する．Obfuscator-LLVM はファイル単位で難読化を適用するため，1,318 個のプログラムそれぞれに対して難読化を行い，その機能維持性，コード変形性を求める．LLVM IR レベルで難読化変換が行われる都合上，Clang 4.0.1 を用いて実行可能コードを生成する．

4.1.3 コード変形性の計測に用いるツール

図 1 の Step. III に該当するコード変形性の算出では，難読化前後のコードから N-gram アセンブリ命令列を求めて，難読化前後のコードのコード間距離を算出する．難読化の適用対象となるコード群の実行可能コードは，評価対象となる難読化ツールごとに対応するコンパイラを用いて生成する．本実験では，アセンブリ命令としてオペコードのみを扱う．オペコードの取得には，GNU Binutils の objdump[2] による線形逆アセンブルを利用し，対象コードの `.text` セクションに含まれるものを取得する．今回の実験では，N-gram の長さ（N）を 1〜3 まで変化させてそれぞれで評価を行う．

4.2 実験結果

4.2.1 機能維持性

実験によって得られた，各難読化方法の機能維持性の一覧を表 3 に示す．表 3 より，Tigress に実装されている難読化方法は，機能維持性が 0.9890〜0.9996 となっ

[2] `https://www.gnu.org/software/binutils/`

表 3: 難読化ツールに実装されている難読化方法ごとの機能維持性

Tigress に実装されている難読化方法					
AddO16	0.9970	EncA_Virt	0.9905	Flat_Virt	0.9890
AddO16_EncA	0.9970	EncL	0.9992	Flat x2	0.9985
AddO16_EncL	0.9970	EncL_AddO16	0.9973	Virt	0.9898
AddO16_Flat	0.9977	EncL_EncA	0.9996	Virt_AddO16	0.9898
AddO16_Virt	0.9909	EncL_Flat	0.9989	Virt_EncA	0.9898
AddO4	0.9970	EncL_Virt	0.9894	Virt_EncL	0.9898
EncA	0.9992	Flat	0.9985	Virt_Flat	0.9901
EncA_AddO16	0.9977	Flat_AddO16	0.9962	Virt x2	0.9898
EncA_EncL	0.9996	Flat_EncA	0.9981		
EncA_Flat	0.9985	Flat_EncL	0.9989		
Obfuscator-LLVM に実装されている難読化方法					
BCF	0.9734	CFF	0.9734	ISub	0.9742
BCF_CFF	0.9727	CFF_BCF	0.9727	ISub_BCF	0.9742
BCF_ISub	0.9734	CFF_ISub	0.9734	ISub_CFF	0.9734

ていることから，元来のコードの機能をほぼ維持して変換できていることがわかる．Virt（コードの仮想化）は，他の難読化方法を組み合わせた場合を含めても 0.9890〜0.9909 となっており，その他の難読化方法と比べると低い値となっている．同様に，Obfuscator-LLVM に実装されている難読化方法は，機能維持性が 0.9727〜0.9742 となっていることから，元来のコードの機能をほぼ維持して変換できることがわかる．

なお，入出力等価性を満たさなかったコード群を調べると，与えられたテストケースによるテストをすべて失敗するようなコードは存在せず，いずれも少なくとも 1 つのテストには成功していたことがわかった．さらに，テストが失敗したコードについて難読化前の状態を調査したところ，確保されていないメモリへの参照があるために，動作（出力）が不安定なコードが含まれていた．難読化対象となるコード群に不具合のあるコードが含まれている場合は，機能維持性の評価に影響を与える可能性があるため，このようなコードを難読化対象から除外するための対策が求められる．この点は今後の課題とする．

4.2.2 コード変形性

実験によって得られた，各難読化方法のコード変形性を図 2 に示す．まず，Tigress の難読化方法のコード変形性に注目すると，Virt や，Virt を組み合わせた難読化のコード変形性は比較的高いことがわかる．Virt を用いた難読化は，機能維持性は比較的低かったものの，コードの内容は大きく変形させる傾向があるといえる．一方，EncL は，1-gram，2-gram，3-gram いずれの場合でも，コード変形性が最も低かった．EncL は，整数や文字列のリテラルを不明瞭にしてそのリテラルを用いているコードを変形するという手法であるために適用対象が大きく限定され，コード内容の変化が小さかったものと考えられる．

Obfuscator-LLVM の難読化方法については，いずれの場合も同程度のコード変形性があることがわかる．Obfuscator-LLVM のほうが Tigress よりもコード変形性が高いのは， Tigress が関数単位での部分的な難読化を適用しているのに対し，ファイルを対象とした難読化を適用しているという違いが影響していると思われる．1-gram でのコード変形性は Tigress の Virt などと大きな違いがないので，使用して

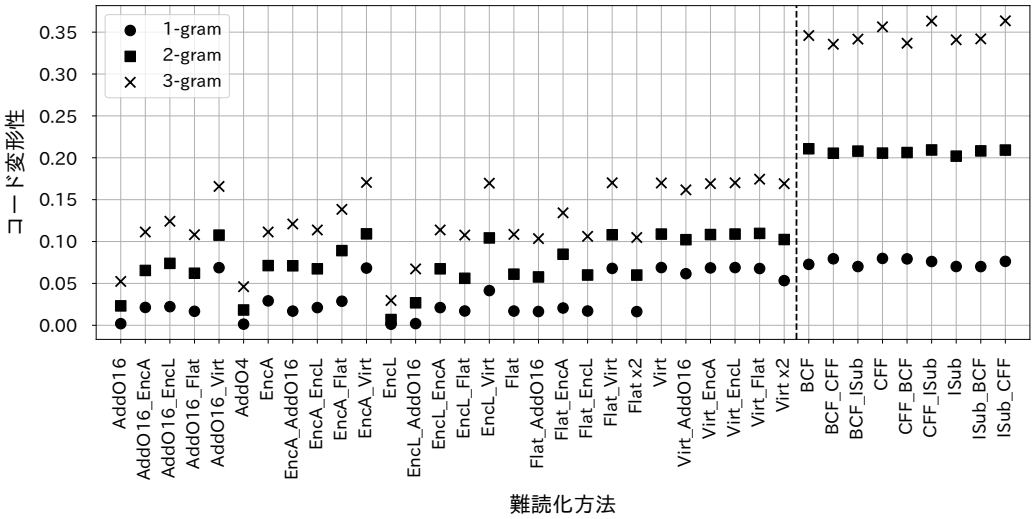

図 2: 難読化ツールに実装されている難読化方法ごとのコード変形性．破線の左側は Tigress，右側は Obfuscator-LLVM.

いる命令の集合の変更度合いという点では大きな違いはないが，2-gram や 3-gram の違いから，難読化後は多様な命令列が使用されていることが分かる．

難読化方法を組み合わせた場合のコード変形性については，その構成要素となる難読化方法のうち高いほうの値に近い結果となっている．つまり，単体で適用した場合と，複数組み合わせて適用した場合どちらについても，コード変形性には大きな差は生じなかった．これは，難読化方法ごとに使用する命令の集合が限られているためであると考えられる．

5 考察

本研究で難読化ツールの信頼性の第 1 の基準と定めた機能維持性については，Tigress の難読化方法はいずれも 0.9890 以上，Obfuscator-LLVM の難読化方法はいずれも 0.9727 以上と高かった．また，第 2 の基準と定めたコード変形性については，Tigress の EncL のようにコード内容の変化が小さいものが存在したものの，ほとんどの難読化方法がコード内容を変化させていることを確認した．どちらの難読化ツールも，大多数のプログラムに対して機能を維持したまま，一定量コードの内容を変化させているという点で，一定の信頼性があるツールであると考えられる．ただし，2 章で述べたように，難読化ツールによって攻撃者の解析が困難になるか，という点については評価を行っているわけではないので，特定の攻撃に対してどれだけ解析が困難になるかの評価が求められる場合は，各攻撃モデルに特化した評価技術を，本フレームワークと組み合わせて利用することが必要である．

本実験では，難読化をするほどコード変形性は上がるが，機能維持性が下がるというトレードオフの関係が示唆されている．機能維持性が 1 でなかったことから，難読化ツールによって不具合が生じる可能性があるという利用者の懸念は実際に正しいことが明らかとなった．難読化ツールの利用者にとっては，難読化したコードに対するテストの実施が必須である．今後，利用者が難読化を適用したいコードと，機能維持に失敗したコードの特徴を比較することにより，難読化の失敗リスクを見積もり，失敗の可能性が高いコードに対するテストを支援するなど，本フレームワークから得られる情報の活用につなげていきたい．

難読化ツールの開発者にとっては，本フレームワークによって機能維持ができな

かったと確認されたコードは，難読化方法およびツールの品質を高めるための重要な手がかりとなると期待している．ただし，今回の実験では，自動生成された多数の C 言語のソースコードを用いたが，その一部には 4.2.1 でも言及したように，動作が不安定なものが含まれていた．現時点では，難読化によって機能が維持されない場合に，それが難読化ツールのバグに起因する（将来修正されうる）のか，難読化方法自体の限界か，あるいは難読化対象のコード自体の問題であるのかを切り分けることが難しい．今後，難読化前の状態においてすでに動作が不安定なコードを除去するなど，評価に悪影響を及ぼす可能性のあるコードを除去するための基準や方法の開発が必要である．

なお，本実験では，それぞれのツールが開発された環境に合わせて，Tigress と Obfuscator-LLVM で異なるコンパイラを用いて実験を実施した．コード変形性はそれぞれのコードごとに測定しているため同一コンパイラ上での効果ではあるが，コンパイラの違いによって難読化方法の効果に差が生じる可能性はありうるため，ツールごとの動作環境の違いを吸収しながら測定条件を均一化することが，技術的な課題の 1 つである．

本研究のフレームワークにおいては，評価プロセスを自動化するために，シンボリック実行を用いている．今回は出力が明確でサイズの小さいコードを対象にしたためシンボリック実行が可能であったが，シンボリック実行は制御構造が複雑なコードに対しては失敗することも多い．そのため，難読化対象の各コードの入力を自動生成できない場合について，ソフトウェアの機能テストからテストケースを収集するなどの対策が求められる．

6 おわりに

本研究では，難読化ツールの信頼性を評価するための方法を検討した．具体的には，難読化ツールの信頼性を判断するためのメトリクスとして，機能維持性およびコード変形性の 2 つの指標を提案し，これらの指標を実証的に評価するためのフレームワークを提案した．本フレームワークは，難読化ツール，難読化対象のコード群とコード群に対応したシンボリック実行を行うスクリプトの 3 つを用意するだけで，難読化ツールに実装されている難読化方法の機能維持性とコード変形性を算出できる．提案したフレームワークを用いた実験では，よく知られた難読化ツールである Tigress と Obfuscator-LLVM の 2 つを対象に機能維持性とコード変形性の評価を行った．どちらの難読化ツールも，実験対象としたプログラムのうちの大多数に対して機能を維持したまま，コードの内容を変化させているという点で，一定の信頼性があることが確認できた．一方で，難読化ツールによって機能が維持されない場合があるという懸念が正しいことも明らかとなった．

今後の課題として，まず，難読化対象とするコードのセットの拡充が挙げられる．オープンソースソフトウェア等で様々なソースコードを収集することは容易であるが，そこから不備のある難読化対象のコードを除外し，除去するための方法の開発や，収集したコードに対する効果的なテストケースの生成あるいは収集方法の検討が必要である．また，難読化ツールにはそれぞれ使用するコンパイラの違いや CPU アーキテクチャの違い，対象プログラミング言語の違いなど，難読化に影響を及ぼす可能性のある要素が多数あることから，ツールの動作環境の違いを吸収しながら測定条件を均一化していく方法を検討していきたい．他の課題として，難読化ツールがコードに与える変化についての評価を，別の角度から行うことが挙げられる．本稿で述べた，Simpson 係数に基づくコード変形性が 0 に近い難読化ツールは，コードに与える変化が小さく，その難読化ツールによって難読化されたコードはオリジナルのコードを予測しやすい傾向にあると判断できる．一方，現状の方法では難読化ツールがコードに与える構造的な変化は十分に把握できないため，特徴的なアセンブリ命令の増加率やソースコードの認知的複雑度 [15] などの指標を用いて，難読

化ツールがコードに与える変化の度合いを異なる角度から評価する方法を検討している．将来的には，難読化ツールの開発者に本フレームワークを利用してもらい，機能維持性およびコード変形性の情報を蓄積，公開することで，難読化ツールの利用者が容易に複数のツールの性能を比較することのできる環境を実現したい．

謝辞 本研究の一部は，JSPS 科研費 JP22K11986，JP22K21279，JP20H05706 および JP19K11916 の助成を受けた．

参考文献

[1] Paolo Falcarin, Christian Collberg, Mikhail Atallah, and Mariusz Jakubowski. Software protection (guest editors' introduction). *IEEE Software, Special Issue on Software Protection*, Vol. 28, No. 2, pp. 24–27, Mar. 2011.

[2] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed application tamper detection via continuous software updates. In *Proc. of the 28th Annual Computer Security Applications Conference*, pp. 319–328, Dec. 2012.

[3] Christian Collberg. The Tigress C obfuscator. `https://tigress.wtf`.

[4] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In *Proc. of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15*, pp. 3–9, May 2015.

[5] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Program Protection.* Addison-Wesley Professional, 2009.

[6] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. A large scale investigation of obfuscation use in Google Play. In *Proc. of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pp. 222–235, Dec. 2018.

[7] Shouki A. Ebad, Abdulbasit A. Darem, and Jemal H. Abawajy. Measuring Software Obfuscation Quality–A Systematic Literature Review. *IEEE Access*, Vol. 9, pp. 99024–99038, Jul. 2021.

[8] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proc. of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, pp. 189–200, Dec. 2016.

[9] Mariano Ceccato, Andrea Capiluppi, Paolo Falcarin, and Cornelia Boldyreff. A large study on the effect of code obfuscation on the quality of Java code. *Empirical Software Engineering*, Vol. 20, No. 6, pp. 1486–1524, Oct. 2015.

[10] Yuichiro Kanzaki, Akito Monden, and Christian Collberg. Code artificiality: A metric for the code stealth based on an n-gram model. In *Proc. of 2015 IEEE/ACM 1st International Workshop on Software Protection*, pp. 31–37, May 2015.

[11] 玉田春昭, 神崎雄一郎. Java バイトコードを対象とした命令の頻度解析による適用難読化ツールの特定. コンピュータセキュリティシンポジウム 2019 論文集, pp. 119–124, Oct. 2019.

[12] 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, Apr. 2019.

[13] Vijaymeena M.K and Kavitha K. A Survey on Similarity Measures in Text Mining. *Machine Learning and Applications: An International Journal*, Vol. 3, No. 1, pp. 19–28, Mar. 2016.

[14] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proc. of IEEE Symposium on Security and Privacy*, pp. 138–157, May 2016.

[15] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability. In *Proc. of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '20, pp. 1–12, Oct. 2020.