マルウェア作成者のコーディング癖の収集法の提案

村上 弘和 [1,2,3,*]　西垣 正勝 [1]

概要：サイバー犯罪の調査に不可欠なインテリジェンス情報は、サイバー攻撃に関連するあらゆる痕跡がソースになる．マルウェアも、サイバー攻撃者が残した重要な痕跡の一つである．インテリジェンス情報の収集を目的としたディープマルウェア解析のうち、マルウェアの筆跡鑑定とも呼べる方法に必要なマルウェア作成者のコーディング癖の情報収集方法を確立することは本研究の 1 つの重要な目的である．マルウェアのコーディング癖の収集法を確立するために、4 つの観点を提案し、それらの観点の留意事項を提示する．具体的な例を用いてマルウェアの筆跡鑑定に利用できるコーディング癖情報がどのようなものかを示す．本稿で述べる観点による抽出法を利用することにより、マルウェア作成者のコーディング癖を比較しやすい形で収集することができる．複数のマルウェアに対しこの方法で抽出した情報を比較分析することにより、マルウェア作成者の同一性、非同一性を示すことができ、サイバー攻撃者に関する一つのインテリジェンスになると考えられる．

キーワード：ディープマルウェア解析，脅威インテリジェンス

The method for collecting coding fingerprints of malware authors

Hirokazu Murakami [1,2,3,*]　Masakatsu Nishigaki [1]

Abstract: All evidence related to cyberattacks is a source of intelligence that is essential for cybercrime investigations. Malware is one of the most important evidence left by cyber attackers. Among deep malware analysis for intelligence collection, one important objective of our research is to establish a method for collecting information on the coding fingerprints of malware authors, which is necessary for a method that can be called malware handwriting analysis. To establish a method for collecting malware coding fingerprints, we propose four perspectives and present notes on these perspectives. Using specific examples to show what coding fingerprints information can be used for malware handwriting analysis. By using this method, the coding fingerprints of malware authors can be collected in a form that is easy to compare. By analyzing some malware with this method, it is possible to show the identity or non-identity of malware authors and it is considered to be one of the intelligence about cyber attackers.

Keywords: Deep malware analysis, Threat intelligence

1. 序　論

マルウェアはサイバー攻撃で使用される方法の一つであり、マルウェアにはサイバー攻撃者に関するインテリジェンスが含まれると考えられる. このインテリジェンスには、マルウェア使用者と作成者の情報が含まれると考えられる [1]. マルウェア解析で見つけることができる情報と出現が予想されるマルウェア作成者・攻撃者像の関係およびレイヤーのイメージを図 1 に示す．この理論が実用できることを示すために、ディープマルウェア解析法の具体的な実施方法を示す必要があると考える．本論文では、図 1 に示した内容のうち、マルウェア作成者のコーディング癖の収集法を提示する(図 1(d)).

コーディング癖が発生する原因は、以下の理由が予想される．

- 「実行結果が同じであれば、あとは作成者の好み次第」というケースでは、使い慣れた方法を選択する可能性が高い．(図 1(d))
- エラーチェックの頻度やその処理など、異常時の対処は作成者の性格や経験などが表れる．(図 1(d))
- コーディングは、「マルウェア作成に関する設計・技術的ポリシー」の影響をある程度受ける．(図 1(b))
- コーディングは、「マルウェア作成者の知識・技能」の影響を大きく受ける．(図 1(c))

実際のプログラミングでは、作成者に任意の方法で任される箇所が存在する．ここに、作成者それぞれの個性が表れる余地がある．また、エラー時の処理は、プロジェクトによってはある程度のルールが決められているものの、通常動作しない処理のため、一般的なソフトウェア開発においてもレビューやテストで見過ごされやすく、統一が不十分となった結果､個性が表れる可能性が考えられる. 一方、アプリケーション作成のプロジェクトでは、設計段階のポリシー設定によりコーディングに一定の制約を受けることがある．また、コーディングは作成者に知識・技能がない

1 静岡大学　創造科学技術大学院
Graduate School of Science and Technology, Shizuoka University
2 東京電機大学　サイバーセキュリティ研究所
Cyber Security Lab, Tokyo Denki University
3 株式会社 CyCraft Japan
CyCraft Japan Corporation.
* murakami.hirokazu.22@shizuoka.ac.jp

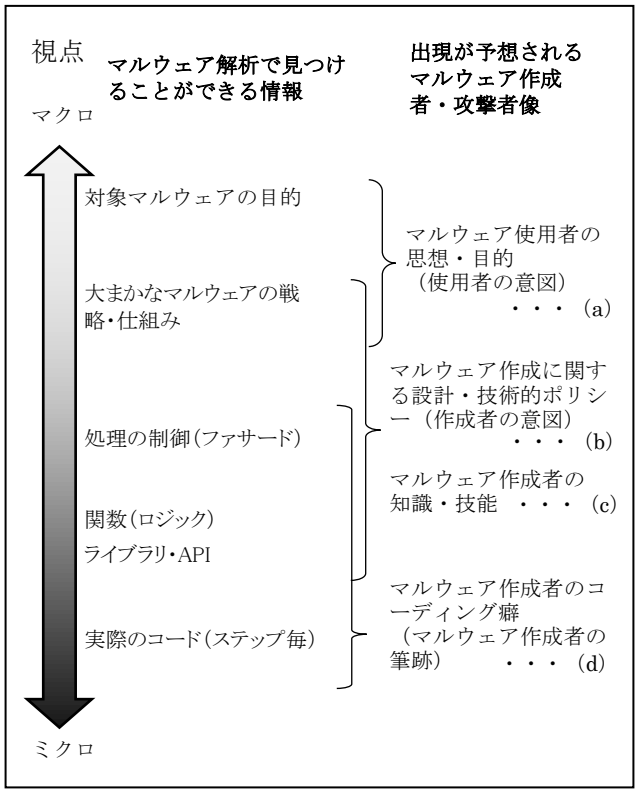


図 1　マルウェア使用者及び作成者の分析のレイヤーのイメージ

ものを実装することはできないほか、知識・技能の不足により冗長または不十分な実装になることがある．図 1 ではコーディング癖は処理の制御(ファサード)よりもミクロな視点であることを示しているが、ファサードやロジックのコーディングやライブラリ・API の利用の特徴はマルウェア作成者の設計・技術的ポリシーや知識・技能の影響も受けることも示している．図 1 の処理の制御(ファサード)以降の特徴は、マルウェア作成者の特徴の表れであり、マルウェアからこれらのコーディング癖をより多く抽出して比較し、合致する比率と内容によって、同一作成者かどうかを判定できるのではないかと考える．それ故に、マルウェアのコーディング癖を比較するためには、観点を定義してマルウェアを解析し、情報を抽出する方法が有用ではないかと考える．

本論文では、実際のマルウェアを用いて試行し、どの程度解析が可能かを検証する．そして、どのような違いが表れ、観点にすべきか調査、検討する．これらにより、サイバー攻撃者のうち、マルウェア作成者の特定に利用できる可能性があること、複数のマルウェアを比較する場合、観点を定義することで異なる時期や解析者によって解析されたとしても、同一項目を比較できるようになることが考えられる．

2. コーディング癖分析のための基本的な原理

コーディング癖の分析のために、4 つの観点を定義する．観点を定義することにより、比較しやすい分類が可能となる．また、観点の違いは解析時のマルウェアの観察方法の違いでもあるため、その違いを明確にするためにも重要である．

2.1 一般的に行われる処理

一般的に、異なるアプリケーションでも、同じ目的の処理を利用することは多い．しかし、同じ目的の処理であっても実装方法に差が出ることがあるため、それを作成者の癖として捉える．癖が表れる原因となるのは、ソースコードの違い、コンパイルやビルドのオプションの違い、開発環境の違いなどが挙げられる．機械語の実行コードの場合、アセンブラによる直接記述の影響を受けることもある．これらの違いが原因で、実行コードに差がでることが考えられる．留意点として、コンパイル済みのライブラリを使用する場合、そのコードが埋め込まれるため、そこでは差が発生しなくなる[2]．その場合でも、そのライブラリが既知のものであれば、「そのライブラリを使う傾向がある」という特徴情報にはなる．

一般的に行われる処理の例は以下のとおり．

- メモリクリア
- メモリのデータコピー
- メモリ領域の確保・開放
- ファイルアクセス
- データ操作（文字列操作、編集、暗号化など）

これらの処理について、差が出る点は以下のとおり．

- 引数チェックの有無および方法
- 引数の型、並び方、有効桁数
- 返り値の内容
- 実装方法（処理内の条件判定、ANSI や Microsoft が定義した関数を利用するか、作成者自身で実装するか）

2.2 エラー処理

エラー判定方法は比較的作成者の癖が表れやすいと考えられる．理由はいくつか挙げられる．第 1 に、「そもそもエラー判定をするか」という考え方そのものに個人差がある．第 2 に、エラーが発生したときのハンドリング方法も、ある程度開発者側に任される．開発規定がある場合でも、その規定に乗せるための処理がある．第 3 に、「目的を達成する必要がある」一般的な処理と違い、通常は実行されないことが前提のため、作成者の取り組み方の性質が異なる．これは、エラー処理をこまめに丁寧に実装している、あるいは軽視して十分なエラー処理を行っていない、といった実装状況に表れる．

エラー処理の内容にも癖が表れやすいと考えられる．エラー処理は、正常時は実行されない．このため、作成者も解析者もエラールートにあまり気をかけず、詳細に分析していないことが多いとみられ、気づかれていない癖が残っていると考えられる．プログラムの作成時のテストでもテストから漏れやすく、異常の発生が困難な場合には机上でのチェックのみとなりやすく、これらの理由により精査されにくい．また、解析者が解析する際にトレースによる方法[3][4]を利用すると、エラー処理が行われないため解析から漏れ、見落とされがちである．また、プログラムでは他者が作成したプログラムを模倣することは多々あるが、エラー処理については見落とされやすいと考えられる．

エラーの判定方法も一律ではない.エラー判定法として、以下に例を挙げる．

- True(1),False(0)
- Success(0),Error(<-1)
- Null(0), そのまま利用する有効なリターン値(>0)
- その他の任意の値の定義

エラー判定方法は様々だが、そもそもエラー判定を行っているか、ということも一つの特徴である．例えば、メモリ領域を確保した場合、有効なアドレスか Null かをチェックすることが一般的であるものの、このチェックを行っていないものも見受けられる．また、判定の結果エラーとなった場合、ハンドリング方法はメッセージの出力の有無、エラー時に処理をアボートするかリトライするか、エラーパラメータの取得を行っているか、といった点で違いが出る．また、Exception の利用の有無という違いもある．これらの特徴は、全て作成者の特徴として利用できるのではないかと考える．

2.3 バグに関する鑑定

通常、マルウェアの作成で意図的にバグを入れることはない．それは攻撃者が意図した動きができない、意図しない動きをするといったリスクになるほか、暴走等によりマルウェアが露見する恐れがあるためである．しかし、マルウェアも人が作ったプログラムである以上、バグが見つかることがある．マルウェアとして正常に動いている以上、影響が無いかごく小さいものであるが、通常は入っているべきではないものであるため、同じバグが複数のマルウェアから見つかった場合、それは同一性を示す大きな特徴となると考える．

マルウェアから見つかるバグの 1 つとして、正しく動くバグがある．これは、処理は作成者が意図したものと違うが、結果的に正しく動くケースである. 発生する例として、整数型の桁数の違いが挙げられる．8 ビットの符号なし整数値のつもりが、32 ビットあるいは 64 ビットで処理されているようなケース[5]である．この場合、結果的には桁あふれなどせず、正常に動作する．他の例として、各種演算処理で、結果的に問題がない場合である．本来の計算方法とは異なるが、結果的に計算結果が合っているような場合である．

マルウェアから見つかる他のバグとして、影響が小さく、具体的に不具合になりにくいバグがある．確保した領域の解放漏れ、領域のサイズ計算に誤りがあるものの、結果は必要サイズより大きくなるため、最終的にメモリ参照エラーにならず問題が発生しない、文字列の末尾に Null 文字を設定する処理が漏れているがゼロクリアされた領域に書き込んでいるため実質的に問題にならない、といったケースである．

2.4 不十分な模倣

マルウェア作成者は、他のマルウェアを模倣することがある．この理由は、同様の目的を達成するため仕組みを参考にすること、マルウェア作成者やマルウェア使用者の容疑を他者に向ける偽旗作戦[6]目的といったことが考えられる．模倣が確認された場合、作成者が異なると判断する指標の一つになると考えられる．模倣の場合、コードは類似するものの、細部が異なることがある．いかに類似していても、ソースが一致し得ないような違いが見つかった場合、模倣を疑う必要がある．それが模倣によるものか、処理の改善のための変更によるものかも考察する必要がある．

仕組みを参考にする場合、処理の流れは似てくるものの、実装方法を全て似せる必要はない．このとき、オリジナルの作成者が意図していた箇所を変更、省略することがある．これは、作成者の考え方の違いや、模倣者の理解不足が原因となると考えられる．

偽旗作戦の場合、他者に疑いを向けるよう既存の別のマルウェアのコードを流用することが考えられるが、そのコード自体は本来のマルウェアでは不要で、解析者に対する目くらましとして導入していることが明らかな場合がある．また、流用元のコードの本当の意図を理解していない場合もある．

3. コーディング癖の判定例

実際の機械語のマルウェアを逆アセンブルしたコードを用いて、各観点でコーディング癖がどのように表れるかを例示する．実際の解析では、例示した方法でマトリックスを使い、比較することが有効である．

3.1 一般的に行われる処理

一般的に行われる処理について、メモリコピー処理についてコードが酷似した例と類似性の低い例を挙げる．コードの酷似した例を図 2 に示す．第 3 引数の値が 0 かどうかのチェック、インデックス値の初期化、ループによって 1

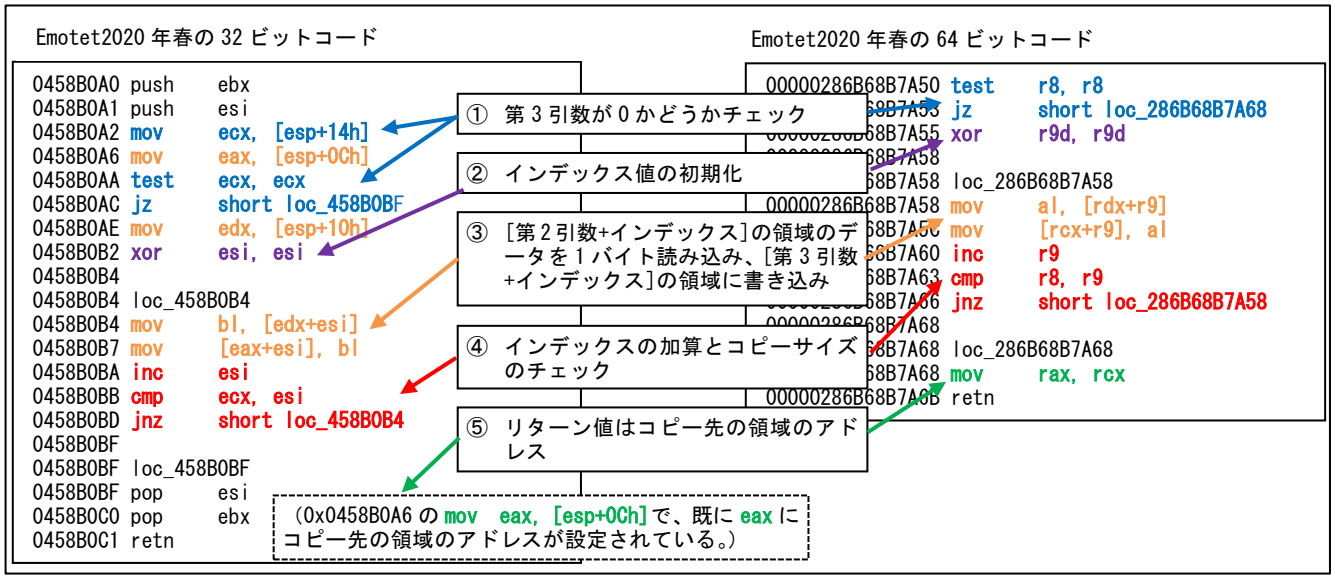


図 2　一般的に行われる処理におけるコードの酷似例

バイトずつコピーする処理、コピーしたサイズのチェック、リターン値の内容といった処理の意味が同じであるほか、使用している命令もほぼ同じで、32 ビット、64 ビットというアーキテクチャーの違いがあってもコードが類似していることが確認できる．コードの類似性の低い例を図 3 を示す．第 3 引数のチェックの有無、コピー処理中におけるインデックス値やコピー元、コピー先の領域の格納方法、リターン値などに違いが見られ、違うコードだと判断できる．特に、引数チェックの有無とリターン値の違いは、ソースコードが違わなければ発生し得ないことが分かる

図 2 および図 3 のコードが酷似した例、類似性の低い例の比較から、ソースが確実に異なることが確実である場合と、そうとは限らないことがあることが確認できた．引数のチェック処理の有無、返り値の設定内容の意味は、コン

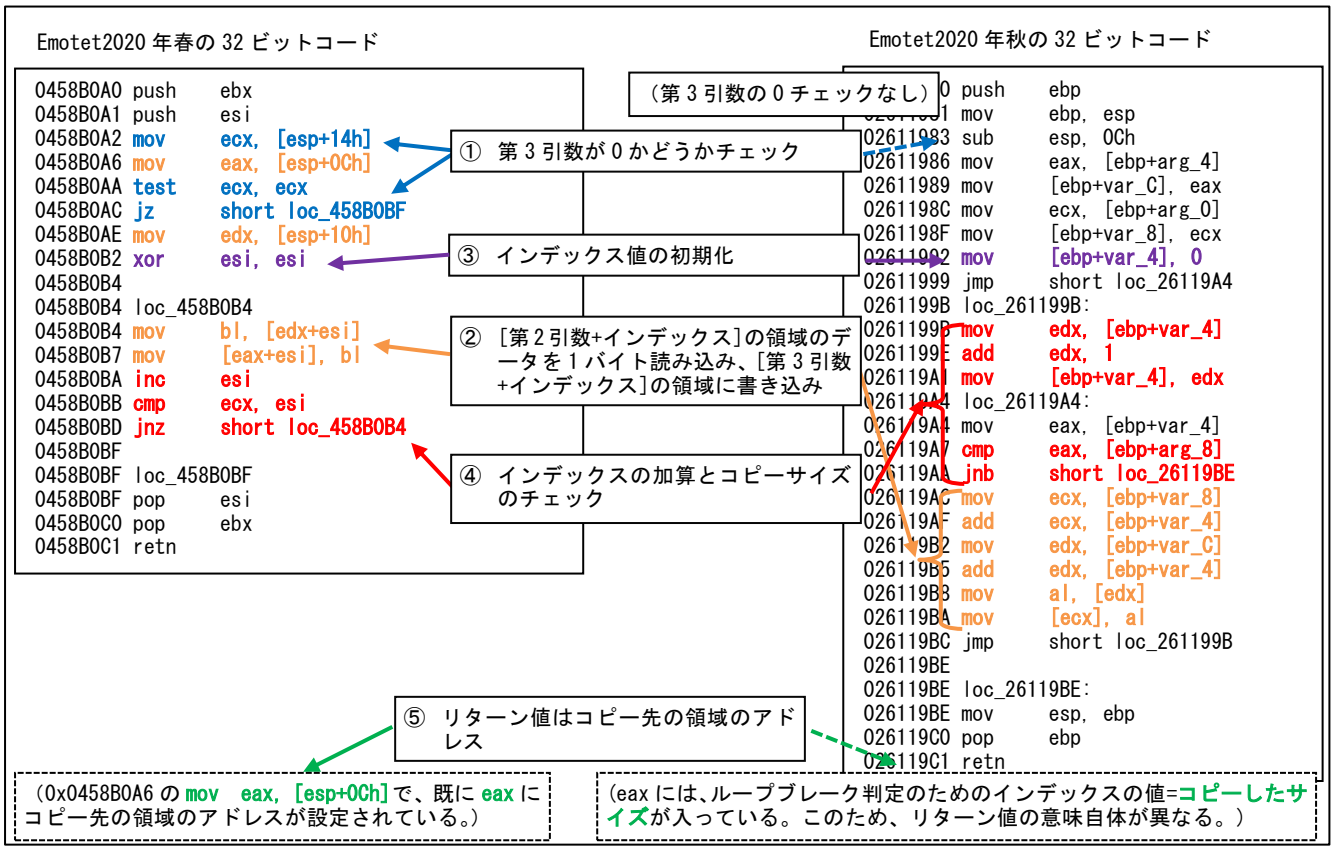


図 3 一般的に行われる処理におけるコードの類似性が低い例

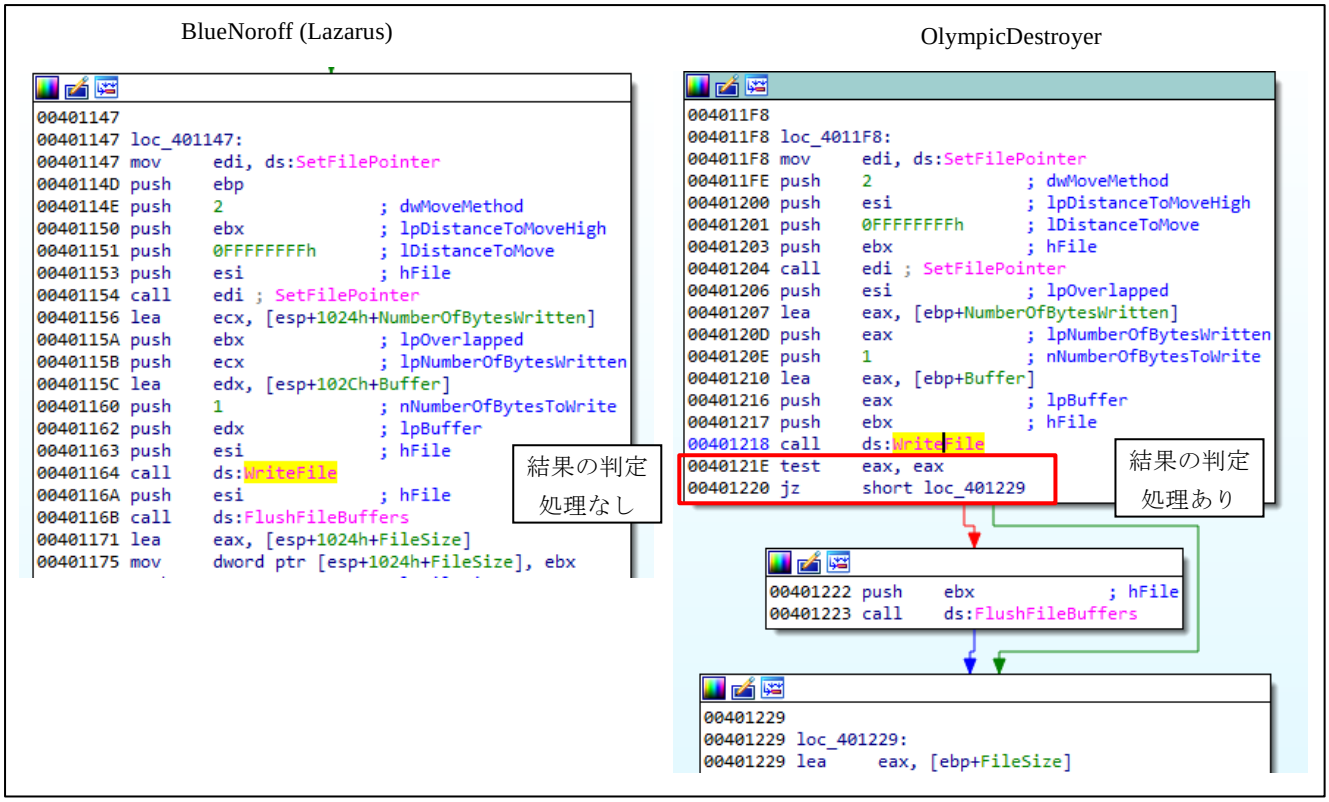


図 4　類似する処理におけるエラー処理の違い

パイラが変更することは考えられず、ソースコードに書かれているはずであり、この違いは強い証拠と考えられる．一方、値を 1 バイトずつ設定方法は変わらないものの、処理が冗長となることで類似性が下がっている．しかし、これは前後の処理も含めたコンパイラの動作（開発環境・オプションの設定）の可能性も考えられるため、必ずしもソースコードが異なるためとは言えないと考えられる．このため、類似性の判定でも重みの違いがあると考える．

3.2　エラー処理

エラー処理は、判定の有無および判定の結果による処理の内容で判定する．エラー処理の違いの例を図 4 に示す．ファイルの書き込みに関する処理に類似性があるが、FlushFileBuffer を実行する際に直前の WriteFile の結果を判定しているか否かという差が表れている．これは、エラー判定の有無によりソースコードに違いがあることを示している．また、図 4 のエラー判定の結果では FlushFileBuffer を実行しない制御のみであるが、エラー処理内で処理の中断やエラーメッセージの出力、リトライなどの処理の違いが表れた場合、やはりソースコードが違うことが確認できる．

3.3　バグに関する鑑定

バグに関する鑑定は、解析において正常でない動作の原因となっているコードを発見し、比較することから始まる．このため、マルウェアの解析においてバグの可能性を常に考慮し、発見した場合にはその箇所、条件、内容を抽出しておく必要がある．抽出内容は、詳細であるほど比較の際に同一性を示す強い特徴量になると考えられる．図 5 は、2 種類の異なるランサムウェアから発見された RC4 を用いたデータの復号処理における変換マップ作製状況である．解析されたマルウェアでは、テキストデータの暗号化・復号化処理の実行のために図 5 の左側のマップを使用している．RC4 では、0x00～0xFF までの 256 個の値のマップを元に 1 バイト(8bits)単位で暗号化・復号化を行うため、マップのサイズは 256 バイトで十分である．しかし、同じマルウェア内でバイナリデータの暗号化・復号化処理のために図 5 の右側のマップを使用している．マップには 0x00～0xFF の 256 個の値を格納しているが、そのデータの単位を 32bit にしていることが確認できる．これは、メモリ領域を 4 倍必要とし、メモリの利用方法としては冗長に見える．この現象の原因を推測すると、左側のマップは#Pragma pack(1)などで明示的に 1 バイト=8bits でコンパイルされたと考えられる．一方、右側のマップは 1 バイト=32bits でコンパイルされたと考えられる．これは、データバスの効率的な利用などの理由により、コンパイラが 1 バイトのデータ幅を拡張することを選択した可能性が考えられる．このケースでは、いずれのマップでも RC4 の暗号化・復号化は正しく動く．したがって、機能的な面ではバグとはいえない．しかしながら、このマルウェア作成者がこのようなマ

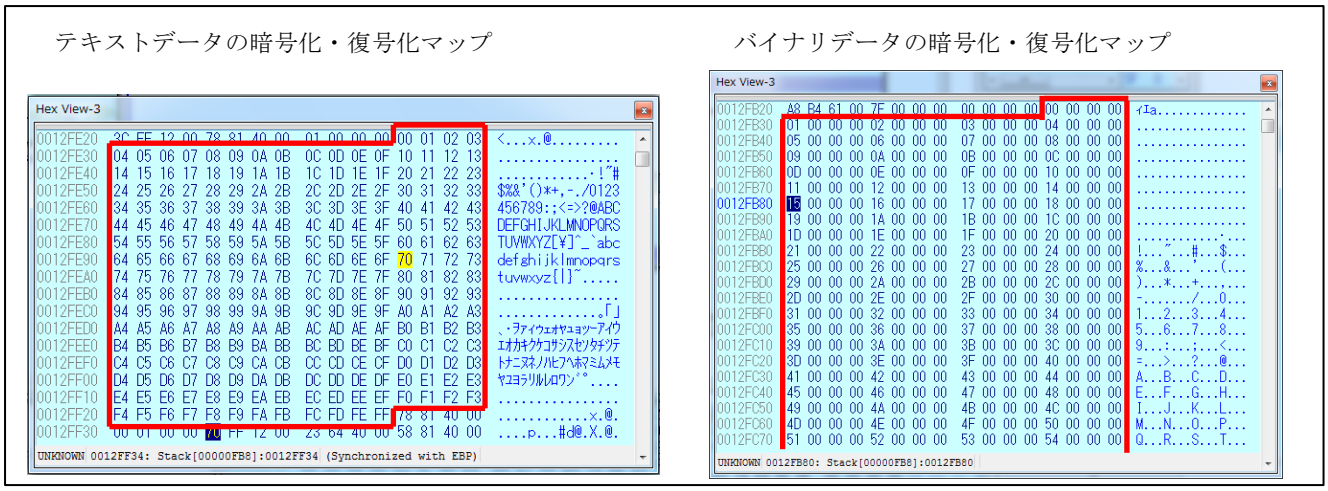
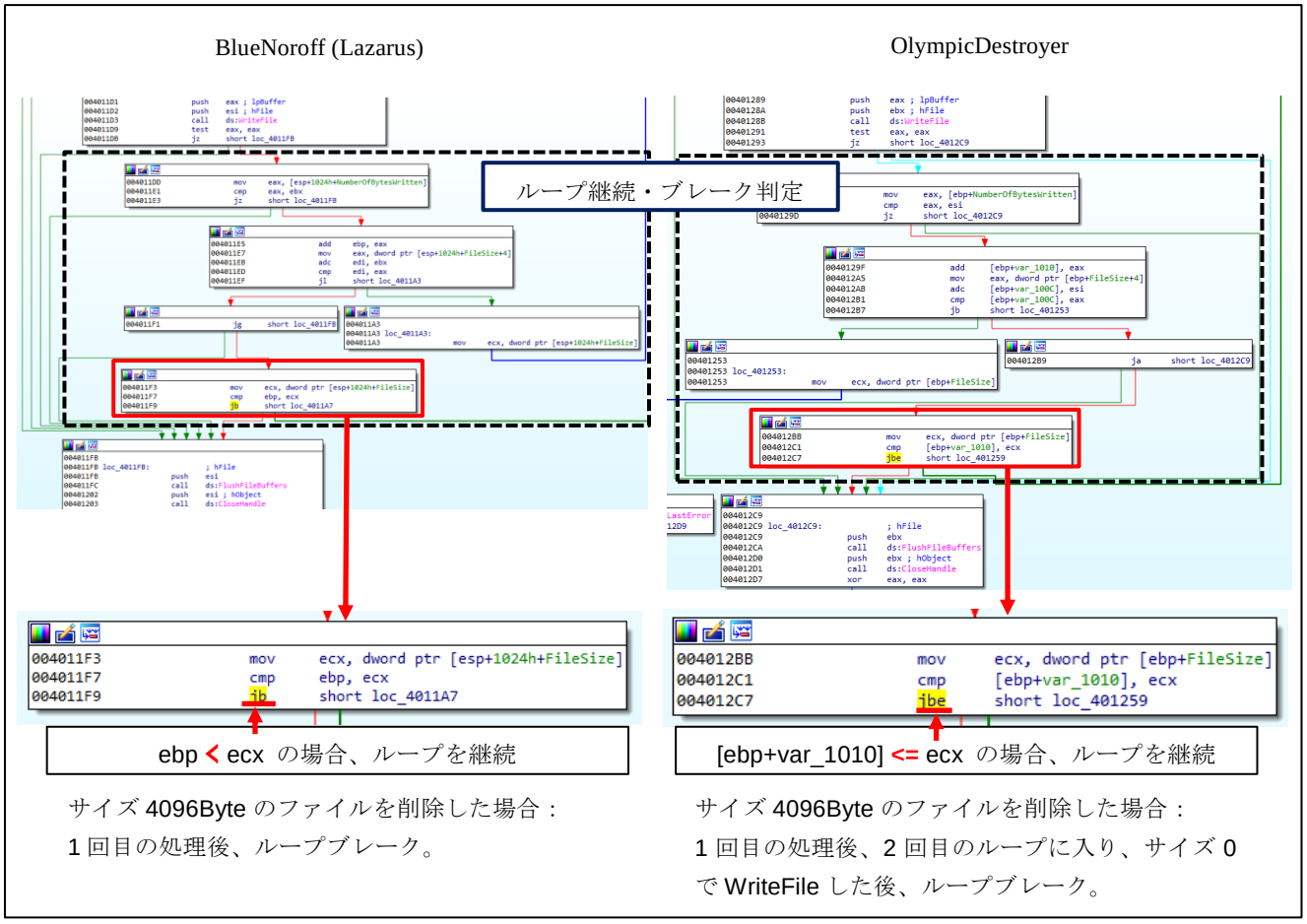


図 5　RC4 におけるマップ作製のバグとみられる現象

ップになることを意図したかというと疑問が残る. 理由は、マップの使い方を分ける合理的な理由が見当たらないためである．このため、マルウェアの作成者の意思とは異なる動きをしているという意味で、バグといえるのではないかと考える．この結果、これは非常に特徴的なコードといえるが、2 つの異なるマルウェアファミリでこれと全く同じ特徴が確認されたケースがあった．これは、これら異なる 2 つのマルウェアを紐づける一つの特徴として利用できると考える.

3.4　不十分な模倣

不十分な模倣は、類似するコードにおいて違うコードである明確な証拠を発見することで判定する．これは、一般的に行われる処理における類似性の否定のケースの延長ともいえる．図 6 は、データの上書きに関する類似した処理の分析結果である．いずれの処理も、ループ処理の 1 回あ



サイズ 4096Byte のファイルを削除した場合：
1 回目の処理後、ループブレーク。

サイズ 4096Byte のファイルを削除した場合：
1 回目の処理後、2 回目のループに入り、サイズ 0 で WriteFile した後、ループブレーク。

図 6　不十分な模倣

たり 0～4096 バイトのサイズのデータで上書きし、さらにデータがあるかを判定して、データがまだあるならばループを繰り返している．しかし、類似する処理ではあるものの、サイズの判定条件が僅かに異なっている．数学の符号で示すと、「<」と「<=」の違いがある．これは、境界値の場合の動作が異なることを示している．仮に、ファイルのサイズが 4096 バイト（もしくはその整数倍）のサイズだった場合、この判定で動作に違いが生じる．BlueNoroff のケースでは、4096 バイトの条件に合致した場合、ループブレークし処理を終了する．しかし、OlympicDestroyer のケースでは、4096 バイトの条件に合致した場合、ループを継続した上で 0 バイトのサイズで WriteFile をした後、ループブレークをしている．コンパイラによってループの実行回数が変化するようなコード出力は考えられず、類似している処理であっても明確にソースコードが違う証拠であるといえる．特に、後に作られた OlympicDestroyer の処理の方が冗長であるため、改善による変化であるとも考えにくい．このため、ファイル消去処理を模倣したものの、完全に模倣できていなかったことが原因であると推測することができる．

4. 結　論

マルウェアの解析では、コードを詳細に分析することでマルウェア作成者のコーディング癖を抽出できると考える．コーディング癖の抽出にあたり、複数のマルウェアの作成者を比較できるようにするために、観点を持って分析してその結果を整理していく必要がある．本論文では、コーディング癖の抽出のための 4 つの観点を提示し、それぞれについての考え方と具体例を示した．今までのマルウェアの解析では、マルウェアであるかどうかの識別方法や機械的なコードの類似性を判定することに重きを置かれていたが、本論文ではサイバー攻撃者のインテリジェンス情報を得るためのディープマルウェア解析の最も基本となる「実際のコードから得られる事実」を抽出し、マルウェア作成者のコーディング癖を判定する方法を示すことができたと考える．この方法を実践し、さらに発展させることにより、マルウェアの筆跡鑑定のような活用が可能になり、サイバー攻撃者のインテリジェンス情報を今以上に充実させることが可能になることによって、サイバー攻撃に対する調査、対策に寄与することに期待したい．

課題として、実際にこの方法を用いて、どの程度作成者の類似性を示せるかを検証する必要がある．また、本論文に示した観点以外にもマルウェア作成者のコーディング癖が抽出できる観点が無いかを研究する必要がある．新たな観点が追加されれば、マルウェアの類似性の判定の正確性が上がると考えられる．

今後の研究では、実際のマルウェアに対しこの観点を用いた分析を行い、マルウェア作成者のコーディング癖の抽出結果に基づいた類似性判定により、マルウェアの筆跡鑑定と呼べる分析ができるかどうかを検証したいと考える．

参考文献

[1] 村上弘和, 西垣正勝, “サイバー攻撃者のインテリジェンス収集のためのディープマルウェア解析”, 2022 年暗号と情報セキュリティシンポジウム(SCIS2022), 2022.

[2] 山口晴広, “ダイナミックリンクとスタティックリンク”, https://atmarkit.itmedia.co.jp/ait/articles/1105/27/news111.html, atmarkIT, 2011.

[3] SANS Institute, “FOR610 Reverse-Engineering Malware:Malware Analysis Tools and Techniques”, 2021.

[4] 大月勇人, “仮想化技術に基づいたマルウェア解析のためのシステムコールトレース手法に関する研究”, https://ci.nii.ac.jp/naid/500001045414, 2016,

[5] “C/C++ のプリプロセッサ リファレンス プラグマ ディレクティブと __pragma キーワード pack pragma”, https://docs.microsoft.com/ja-jp/cpp/preprocessor/pack?view=msvc-170, Microsoft, 2022.

[6] “The devil’s in the Rich header”, https://securelist.com/the-devils-in-the-rich-header/84348/, SECURELIST, 2018.