解析済みマルウェアとの差分抽出による静的解析の効率化手法の提案

羽田 大樹† 後藤 厚宏†

†情報セキュリティ大学院大学
221-0835 神奈川県横浜市神奈川区鶴屋町 2-14-1
{mgs115505, goto}@iisec.ac.jp

あらまし 社会インフラにおけるマルウェアの被害においては，フォレンジックによる正確な被害範囲の特定が必要となる．このような状況におけるマルウェア解析では，手動での静的解析による厳密な解析作業が必要である．本稿では，既に解析が完了した複数のマルウェアの情報を用いて，効率的に静的解析を行うアーキテクチャを提案する．本アーキテクチャでは，プログラムをグラフ構造で表現して2つのプログラムの差分を特定する手法を活用する．実際のマルウェア検体を用いてこの提案アーキテクチャを評価し，解析が完了した複数のマルウェアと対象とするマルウェアとの差分抽出が静的解析の効率化に有効となる事例を示す．

Extracting Differences Among the Same Kind of the Malwares to Make Static Analysis Efficient

Hiroki Hada† Atsuhiro Goto†

†Institute of Information Security
2-14-1 Tsuruya-cho, Kanagawa-ward, Yokohama 221-0835, JAPAN
{mgs115505, goto}@iisec.ac.jp

Abstract It is crucial to analyze malware behaviors precisely and efficiently to clarify the affected extent in the forensic process when our social infrastructure systems are under targeted-attack. In this research, we propose the new analysis system architecture which makes static malware analysis effective. One of the key components in this system is a similarity analysis function which compares execution code of the target malware with already known malware in the database. We evaluated this component with some malware samples and show this architecture is available.

1. はじめに

政府や企業におけるマルウェアの脅威がますます増大している．アンチウイルスソフトウェアの検知を回避するための技術の進化やマルウェアに感染させるよう誘導する手口の巧妙化といった事が背景として挙げられる．

アンチウイルスソフトウェアの検知率が低下している最大の理由は，マルウェアの亜種が大量生成される事である．簡単な操作でマルウェアを作成できる開発ツールや，同じ機能を持つ別のマルウェアを生成する変換ツールが存在し，知識がなくともマルウェアの亜種を簡単に作成する事ができる．またポリモ

ーフィック型マルウェアやメタモーフィック型マルウェアなど，拡散する際に自身を改変する機能を持つものも存在する[1]．マルウェアは高レベル言語で開発され，作者が頻繁にアップデートを行っているケースが多い事も理由である．

近年は APT (Advanced Persistent Threat)や標的型攻撃といったキーワードに代表されるように，軍事機関，政府，大企業に対して高度かつ目的を持った攻撃が行われ，被害の大規模化や深刻化が問題となっている．このような攻撃は慎重に行われるため，攻撃者は自分のマルウェアがアンチウイルスソフトウェアで検知されない事を確認してから攻撃に利用する．情報資産をこのような攻撃から完全に守る事を想定するのは現実的ではなく，また被害を受けた場合はシステムを復旧するとともにフォレンジック調査を行い侵入経路や影響範囲を正確に特定する事が重要となる．

マルウェア解析の一般的な手法は，大きく動的解析と静的解析の 2 種類に分類できる．動的解析では，マルウェアを実行してシステムコールや API の呼び出し，ファイルやレジストリへの変更，ネットワーク通信などを観測する事でマルウェアの挙動を解析する．効率的に解析できる一方で，解析されている事を検知して動作を停止するマルウェア，ボットのように攻撃者からの指令で動作するマルウェア，特定の時刻にのみ動作するマルウェアに関して，その挙動を完全に抽出する事は難しい．静的解析では，動的解析では調査できない挙動についてアセンブリを直接読み解く作業を行う．原理的には完全に解析する事が可能であるが，技術者のスキルと多大な時間を要するという問題がある．

フォレンジック調査のように厳密に挙動を解析する必要がある状況では，動的解析だけでなく静的解析を組み合わせて調査する必要があり，今後このような状況はますます増えていくと考えられる．

本研究では，マルウェアの挙動を厳密に特定する必要がある状況において，既に解析が完了した複数のマルウェアの情報を用いて効率的に静的解析を行うためのアーキテクチャを提案する．このアーキテクチャでは，最初に解析対象マルウェア検体のアンパックと逆アセンブルを行い，マルウェアデータベースから類似するマルウェアを検索する．その後，プログラムをグラフ構造で表現して 2 つのプログラムの差分比較を行う手法を利用して，解析対象マルウェア検体における解析対象部分を限定する．また，実際に存在するマルウェアを用いてこの提案アーキテクチャを評価し，解析が完了した複数のマルウェアと対象とするマルウェアとの差分抽出が，静的解析の効率化に有効となる事例を示す．

2. 関連研究

2.1. マルウェア分類

マルウェアの解析は技術的に高度かつ時間のかかる作業であるため，解析作業を効率化するための様々な研究が行われている．その中のテーマのひとつであるマルウェア分類では，多数のマルウェアに対して同種の機能や構造を持つマルウェアを同じグループとして自動で分類する手法が提案されている．分類を行う事で，検体と類似するマルウェアが既に解析済みである場合，マルウェアの動作や構造をある程度推定できるため，解析作業者の負担を減らす事ができる．

マルウェア分類を実現するためには 2 つのマルウェアが類似するかどうかを判定する指標が必要である．この指標として様々な提案がされているが，大きく分けて逆アセンブリのコード列の並びに着目した手法と，プログラム構造に着目した手法に分類される．

逆アセンブリのコード列の並びに着目した手法として，岩村らは逆アセンブリのコード列の最長共通部分列（LCS）の長さを用いて類似度を定義する手法を提案している[2]．Md. E. Karim らは n-gram と n-perm という指標を用いた分類手法を提案している[3]．M. Gheorghescu は，ベーシックブロックを単位

として 2 つのマルウェア間のレーベンシュタイン距離を算出して非類似度とする手法を提案している[4].

これらの手法は逆アセンブリのコード列という機械語レベルの情報に基づいて判定できる一方で，コンパイラやコンパイルの最適化オプションの変更等で同じソースコードから全く異なる実行コードを出力した場合，これを類似するマルウェアと判定する事が難しい．このような状況に対応するため，プログラム構造に着目して，制御構造をグラフで表現する事でプログラムの特徴を比較する手法がある．岩本らは，共通のソースコードから作成されたマルウェアは API 関数の呼び出し順序も変わらない事に着目し，制御フローグラフから API 推移を抽出してクラスタ分析を行い分類する手法を提案している[5].

2.2. アンパックと逆アセンブル

ほとんどのマルウェアでは，圧縮や暗号化などの手法を用いて自身の実行コードを変換して隠蔽するパッキングという処理が行われている．このため，オリジナルコードの逆アセンブリを取得するには技術が必要である．この作業を自動化するために，M. G. Kang らは書き込みが発生したメモリ領域がその後実行された場合に当該領域をオリジナルコードとして出力する手法を提案している[6]．川古谷らは，アンパックの展開ルーチンからパッカーの種類を特定する手法を提案している[7]．また，岩村らはコンパイラ出力コードの尤度に基づいて機械語の逆アセンブルの精度を向上する手法を提案している[8].

2.3. プログラムの差分比較

2.1 のマルウェア分類に関する研究では，2 つのマルウェアの類似度を高速に算出する事が目的であり，類似度を計算するために重要ではない情報を削減していた．そのためマルウェアの差分を特定する事は直接的には行っていなかった.

リバースエンジニアリングの分野において，コードの盗用の特定や，ソフトウェアのセキュリティパッチによる変更箇所の特定などを目的として，2 つのプログラムを比較して差分を出力するための研究が行われている．H. Flake は, プログラムの逆アセンブリのコード列を制御命令単位で分割し，関数の呼び出し関係をグラフで表現したコールグラフと，その他の制御命令によるジャンプをグラフで表現した制御フローグラフとしてグラフの入れ子構造で表現して，コールグラフのノードのマッチングを行う事で 2 つのプログラムの差分を特定する手法を提案している[9]．コールグラフの各ノードは (ベーシックブロックの数, ノード内の辺の数, ノード間の辺の数) という 3 次元座標の特徴量が定義される.

T. Dullien らはさらにこの手法を拡張し，Property 関数を用いてノード集合を適切に分割する事で精度を向上させる手法を提案し，Microsoft Windows のセキュリティ更新プログラムの修正内容を特定している[10].

3. 提案方式

本研究では，既に解析が完了した類似マルウェアを用いて解析対象マルウェア検体の解析範囲を限定して作業を効率化する図 1 のアーキテクチャを提案する．本アーキテクチャは以下の①～④の手順で実行される.

手順① マルウェア感染 PC からマルウェア検体を特定し，マルウェア検体のアンパックと逆アセンブルを行う.

手順② 解析対象とするマルウェア検体と類似する解析済みのマルウェアをマルウェアデータベースから検索する.

手順③ 解析対象マルウェア検体と，解析済みの類似マルウェアとの差分を抽出して，解析作業者に解析対象マルウェア検体の未解析部分のみを渡す.

手順④ 解析作業を行い，解析結果を逆アセンブリにコメントとして付与し，データベースにフィードバックする.

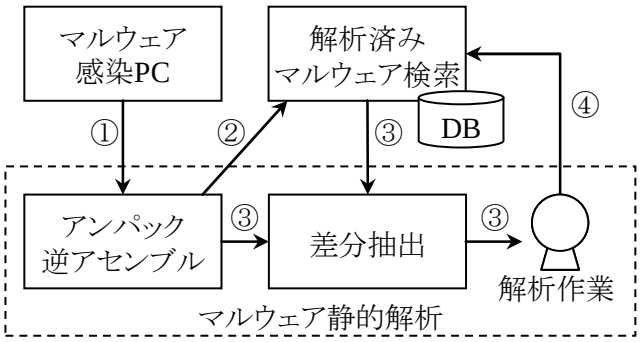


図 1 提案アーキテクチャ

本システムの提案にあたり，以下の条件を想定している.

1. 解析済みマルウェアデータベースが存在し, 入力したマルウェア検体の情報をもとに類似する解析済みマルウェアをデータベースから検索して, 解析データとともに出力可能である.
2. 入力とする 2 つのマルウェアのアンパックと逆アセンブルを行うシステムが存在する.
3. 本アーキテクチャは解析作業者の作業を効率化するための支援を目的としている. 最終的には手動で静的解析を行う.

手順③で利用する差分抽出システムは本アーキテクチャを実現するための重要な構成要素である．ここでは 2 つのマルウェアの差分を抽出する方式として，プログラム構造に着目した H. Flake の方式を用いる.

4. 評価

本アーキテクチャの有効性について，マルウェア検体提供サービス[12]から入手した複数のマルウェア検体を対象に評価を行った. 差分を抽出する方式には，グラフを利用してマルウェアの構造を比較する方式を採用するため，H. Flake の方式を実装したソフトウェアである BinDiff[11]を用いた．BinDiff は，逆アセンブラ IDA Pro[13]が出力した逆アセンブリとコールグラフを入力として，2 つのコールグラフにおける関数の対応関係を出力する．さらに，対応する関数同士を比較してベーシックブロックの対応関係を出力する.

検体が持つ関数のうち, 既に解析されたマルウェアと比較して関数がカバーできる割合を今回の実験における精度の指標として評価する. ただし, BinDiff は関数を比較する際にその一致率を 0 から 1 の範囲で出力するが, 図 2 と図 3 の比較結果の通り, 一致率の低いものについては関数の構造が対応していたとしても解析作業量の削減にはならない. このため, 今回は一致率が 0.5 以上の関数のみを対応していると見なして評価を行った.

4.1. 処理がほぼ一致する事例

表 1 の W32.Morto と呼ばれる 2 つのマルウェア検体について実験を行った結果，表 2 の通りほとんどの関数は共通し，コールグラフがほぼ同型である事が示せた.

表 1 マルウェア検体（W32.Morto）

検体	ハッシュ値（MD5）
検体 1	4F0503566E1021D69B1578EF1882F076
検体 2	39E4F9B500ADD54D894E39BBE9A47302

表 2 検体 1，2 における比較結果

関数	個数
検体 1，2 に共通	211
検体 1 にのみ存在	0
検体 2 にのみ存在	3

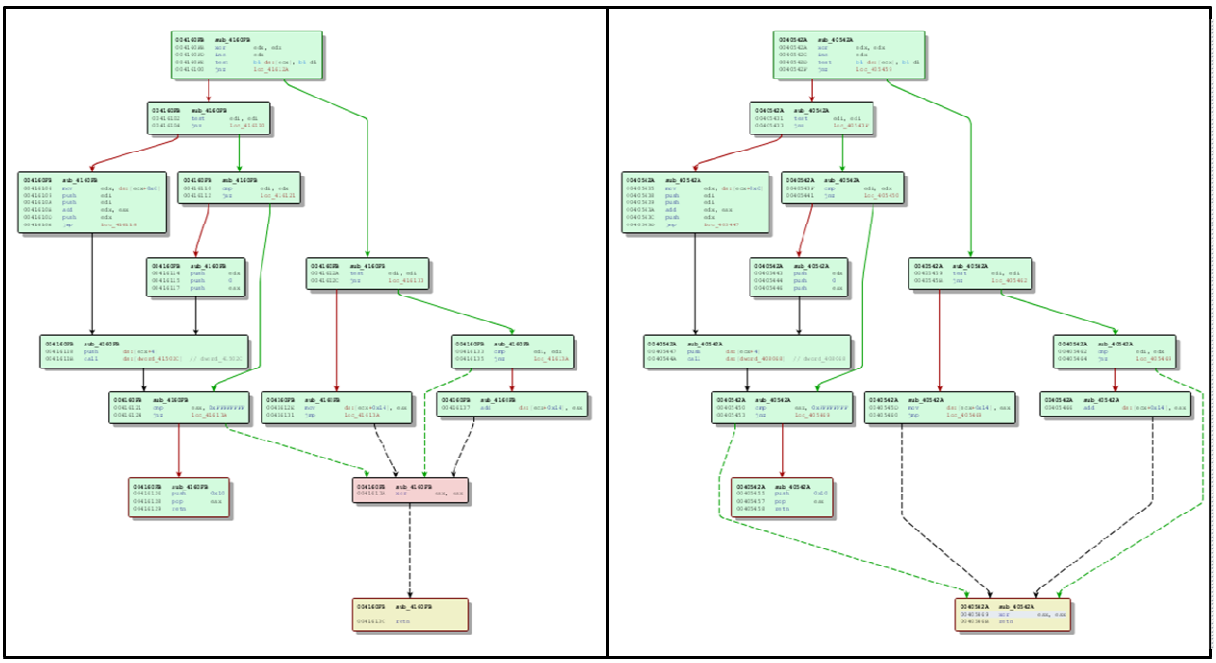


図 2 一致率 0.90 の関数の比較結果の例

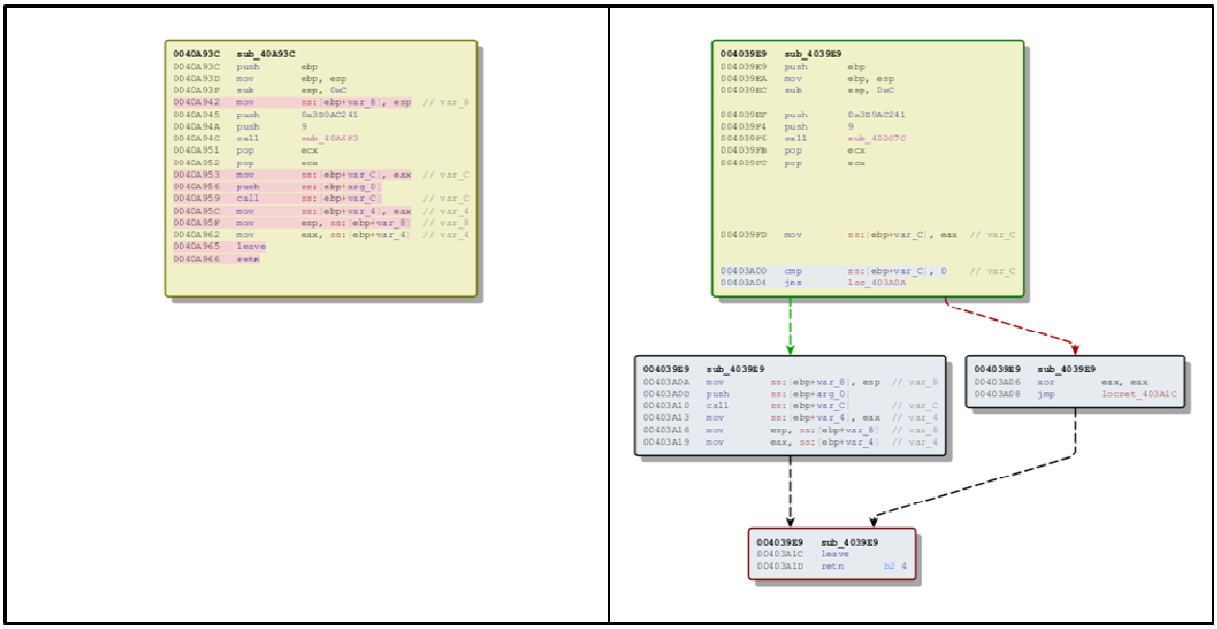
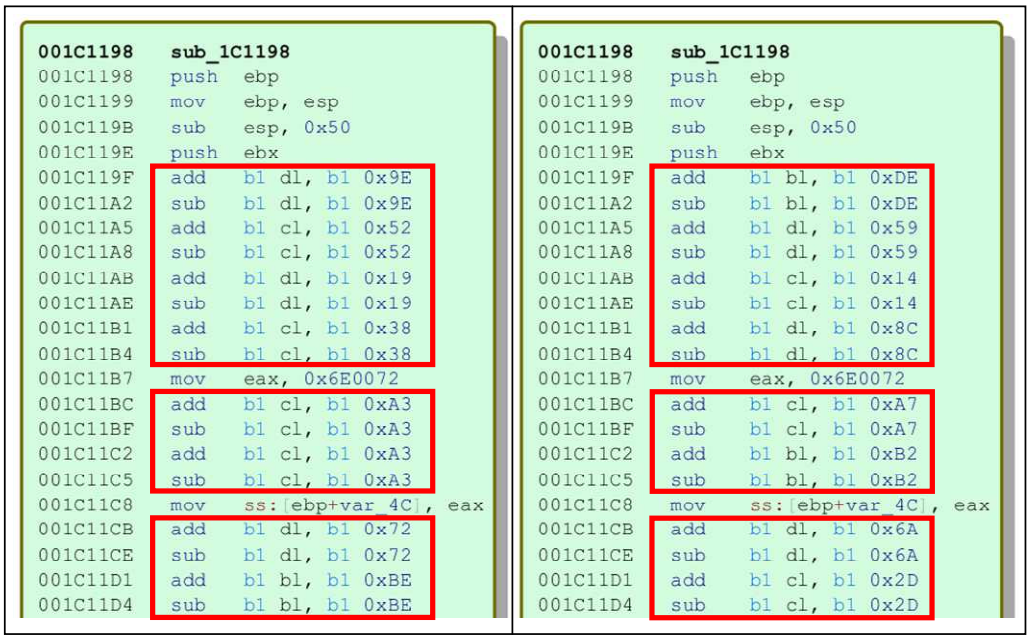


図 3 一致率 0.21 の関数の比較結果の例

検体 1, 2 に共通する関数においても内部の処理が完全に一致するとは限らないため個々に調査する必要があるが，関数の対応関係とその内部のベーシックブロックの差分を特定した事で, その検体の解析作業が削減される. 本検体において関数内部のベーシックブロック内で変化があった部分は全て図 4 のような難読化のための冗長な処理に関してのみであった．適当なレジスタに対して固定値を加算してから同じ値を減算する事で，プログラムの動作に影響しない処理を追加しているが，マルウェアの亜種毎にこの固定値を変化させているだけであり，動作には変化がない事を解析者が容易に確認する事ができた.



```
001C1198   sub_1C1198
001C1198   push   ebp
001C1199   mov    ebp, esp
001C119B   sub    esp, 0x50
001C119E   push   ebx
001C119F   add    bl dl, bl 0x9E
001C11A2   sub    bl dl, bl 0x9E
001C11A5   add    bl cl, bl 0x52
001C11A8   sub    bl cl, bl 0x52
001C11AB   add    bl dl, bl 0x19
001C11AE   sub    bl dl, bl 0x19
001C11B1   add    bl cl, bl 0x38
001C11B4   sub    bl cl, bl 0x38
001C11B7   mov    eax, 0x6E0072
001C11BC   add    bl cl, bl 0xA3
001C11BF   sub    bl cl, bl 0xA3
001C11C2   add    bl cl, bl 0xA3
001C11C5   sub    bl cl, bl 0xA3
001C11C8   mov    ss:[ebp+var_4C], eax
001C11CB   add    bl dl, bl 0x72
001C11CE   sub    bl dl, bl 0x72
001C11D1   add    bl bl, bl 0xBE
001C11D4   sub    bl bl, bl 0xBE
```

```
001C1198   sub_1C1198
001C1198   push   ebp
001C1199   mov    ebp, esp
001C119B   sub    esp, 0x50
001C119E   push   ebx
001C119F   add    bl bl, bl 0xDE
001C11A2   sub    bl bl, bl 0xDE
001C11A5   add    bl dl, bl 0x59
001C11A8   sub    bl dl, bl 0x59
001C11AB   add    bl cl, bl 0x14
001C11AE   sub    bl cl, bl 0x14
001C11B1   add    bl dl, bl 0x8C
001C11B4   sub    bl dl, bl 0x8C
001C11B7   mov    eax, 0x6E0072
001C11BC   add    bl cl, bl 0xA7
001C11BF   sub    bl cl, bl 0xA7
001C11C2   add    bl bl, bl 0xB2
001C11C5   sub    bl bl, bl 0xB2
001C11C8   mov    ss:[ebp+var_4C], eax
001C11CB   add    bl dl, bl 0x6A
001C11CE   sub    bl dl, bl 0x6A
001C11D1   add    bl cl, bl 0x2D
001C11D4   sub    bl cl, bl 0x2D
```

図 4 検体 1，2 のマルウェアの差分

4.2. 差分が抽出できる事例

表 3 の SpyEye と呼ばれるマルウェア検体について実験を行った．ただし本検体についてはパッキングが行われていたため手動でアンパックを行った．ここで，検体 3 はこれから解析するマルウェア検体とし，検体 4，5，6 はデータベースに格納された解析が完了したマルウェアである事を想定した．検体 3，4 を比較した結果を表 4 に示す．

表 3 マルウェア検体（SpyEye）

検体	関数	ハッシュ値（MD5）	想定する役割
検体 3	523	9D2A48BE1A553984A4FDA1A88ED4F8EE	解析対象マルウェア検体
検体 4	139	D64CA15261C53279A7288616B3CB1A92	解析済みマルウェア
検体 5	609	DF04C2CD2B5F7E471CB0435FDB9B3014	解析済みマルウェア
検体 6	218	42DACFBE2E5AF0C43D17356CA76F0271	解析済みマルウェア

表 4 検体 3，4 における比較結果

関数	個数
検体 3，4 に共通	53
検体 3 にのみ存在	389
検体 4 にのみ存在	5

この結果，このサンプルでは 53 / 523 = 10.1% の関数が共通し，この部分が解析作業削減のために利用できる事が示せた．

4.3. 複数の類似マルウェアとの比較が有効となる事例

解析済み類似マルウェアが複数ある場合，これらを合わせて比較する事で解析対象とするマルウェアの未解析部分をさらに削減できる場合がある．解析対象マルウェア検体と比較する解析済みマルウェアとして，検体 4 に加えて検体 5, 6 を追加して実験を行った結果を表 3 に示す．

表 3 検体 3 と検体 4, 5, 6 の比較結果

	検体 3 vs 検体 4	検体 3 vs 検体 5	検体 3 vs 検体 6	検体 3 vs 検体 4, 5, 6
共通する関数	53	78	85	135

検体 4，5，6 それぞれ単体と比較した場合と比べて，解析対象とするマルウェア検体の解析済み関数の数が増加した．3 つの類似マ

ルウェアを利用する事で解析作業削減のために利用できる関数の数を 135 / 523 = 25.8% まで向上させられる事が確認できた.

5. まとめと今後の課題

大量に作成されるマルウェアの亜種を広く網羅的に把握できるよう，近年のマルウェア解析の研究テーマとしては，いかに作業を自動化して高速かつ大量に処理できるかを考える事が主流である．一方で我々は，重要インフラが狙われる今日ではフォレンジックにおいて特定のマルウェアに焦点を当てた厳密な解析がこれまで以上に重要になると考えている．そのため，解析作業の完全な自動化を目指すのではなく，熟練した技術者が行う解析作業を効率化するアーキテクチャを提案して評価を行い，効率化できる事例を示した.

今回の実験ではグラフ構造を用いた差分抽出の方式により関数の一致率が高い事例を発見する事で今回の方式が有効である事を示したが，関数の一致率が低い場合において，マルウェアの挙動が全く異なるために一致率が低い結果となったのか，マルウェアの挙動が同じにも関わらず一致率が低い結果になったのかを区別する評価は行っていなかった．更新プログラムのリバースエンジニアリングの場合と異なり，マルウェア解析の場合は意図的に不要なコードを埋め込まれる場合があるため，今後はコンパイラのオプティマイザ，パッカー，難読化ツールの振る舞いを調査しながら，データフロー解析等を用いて関数やベーシックブロック毎の差分抽出の精度を向上させたい.

また，我々の提案方式では，マルウェアの分類，アンパック，逆アセンブルが高い精度で実現可能であるという事を前提としており，進化するマルウェアにも対応できるようこれらの技術も継続して改良を行う必要がある.

参考文献

[1] P. Szor, P. Ferrie. Hunting for Metamorphic. Virus Bulletin Conference, 2001.

[2] 岩村誠, 伊藤光恭, 村岡洋一. 機械語命令列の類似性に基づく自動マルウェア分類システム. 情報処理学会論文誌 Vol. 51 No. 9 1-11, 2010.

[3] Md. E. Karim, A. Walenstein, A. Lakhotia, L. Parida. Malware phylogeny generation using permutations of code. European Research Journal of Computer Virology 1, 1-2 (Nov. 2005) 13-23.

[4] M. Gheorghescu. An automated virus classification system. In Virus Bulletin Conference, October 2005.

[5] 岩本一樹, 和崎克己. 静的解析によるマルウェアの分類と結果の検討. DICOMO, 2010.

[6] M. G. Kang, P. Poosankam, H. Yin. Renovo: a hidden code extractor for packed executables. In Proc. WORM' 07: Proceedings of the 2007 ACM workshop on Recurring malcode, 2007, pp. 46-53.

[7] 川古谷裕平, 岩村誠, 針生剛男. 実行命令トレースに基づく動的パッカー特定手法. 情報処理学会 CSS, 2011.

[8] 岩村誠, 他. コンパイラ出力コードの尤度に基づくアンパッキング手法. 情報処理学会 CSS, 2008.

[9] H. Flake. Structural Comparison of Executable Objects. DIMVA, 2004.

[10] T. Dullien, R. Rolles. Graph-based comparison of Executable Objects. SSTIC, 2005.

[11] Zynamics BinDiff. http://www.zynamics.com/bindiff.html.

[12] Offensive Computing. http://www.offensivecomputing.net.

[13] IDA Pro. http://www.hex-rays.com.