Vol.2016-CSEC-73 No.5
Vol.2016-IOT-33 No.5
2016/5/26

BinGrep: 制御フローグラフの比較を用いた関数の検索による マルウェア解析の効率化の提案

羽田 大樹[†1 †2]　後藤 厚宏[†1]

概要：近年，日本においても広範囲な APT 攻撃による大規模な被害を経験した．明確な目的を持って行われる高度かつ執拗な攻撃においてはその被害も甚大となり，事後の対応も迅速かつ確実な判断が要求される．このようなインシデント対応では，まずネットワークや端末のログから確実に被害範囲を特定することが求められるが，ここでマルウェアの接続先の URL や暗号アルゴリズムなどが重要な情報となる．これらの情報をマルウェア解析で調査する場合，調べたい処理を行う実行コードの場所を特定できると，作業者は速やかに解析にとりかかることができる．本研究では，インシデント対応におけるマルウェアの静的解析を効率化するため，過去に調査したことのあるマルウェアの関数を入力として，制御フローグラフの編集距離を利用することで，解析するマルウェアにおける該当の関数を検索するアルゴリズムを提案する．正常プログラムによる評価では GNU bash と GNU binutils の 11049 個の関数の 90.3% について，マルウェアによる評価では 20 個の関数の 95.0% について，上位 10 位以内に正しく正解を出力できることを示した．

キーワード：マルウェア解析，静的解析，フォレンジック

BinGrep: Proposing the efficient static analysis method by searching for the function comparing control flow graphs

Hiroki Hada[†1 †2]　Atsuhiro Goto[†1]

Abstract: In recent years, many Japanese organizations experienced a large-scale damage caused by APT activities. The damage of advanced and persistent attack is serious, and prompt and appropriate incident response is required. In such an incident response, the information such as malicious URL destination and cryptographic algorithms used by malware are important to identify the effect of the incident. When a malware analyst wants to analyze these information, identifying the position of specific function code is useful to get started immediately. In this paper, to improve the efficiency of the static analysis in incident response, we propose function searching algorithm that employs edit distance of the control flow graph and uses malware investigated before. We evaluated that 90.3% of the 11049 functions of the GNU bash and GNU binutils as normal program and 95.0% of the 20 functions as malware can be output correctly within top 10.

Keywords: Malware Analysis, Static Analysis, Forensic

1. はじめに

近年, 日本においても広範囲な APT 攻撃による大規模な被害を経験した．カスペルスキー社の報告では，日本企業の 300 社以上が被害にあっているとされる[1]．政府や重要インフラだけでなく日本の様々な業種における数百単位の大規模組織をターゲットに攻撃が行われ，共通的に Emdivi という RAT 型マルウェアを使用していた．JPCERT/CC によると, この一連のキャンペーンは標的型攻撃から始まり，脆弱性やパスワードリストなどを駆使してシステムの深くまで侵入し大量の機密情報や個人情報を収集していた[2][3].

明確な目的の下で行われる高度かつ執拗な攻撃においてはその被害も甚大となり，復旧や感染経路の特定，影響範囲の調査など事後の対応についても慎重かつ確実な判断が要求される．このようなインシデント対応においては，しばしばフォレンジックが必要となる．NIST では，フォレンジックを「収集」「検査」「分析」「報告」という 4 つの工程で説明している[4].「収集」では，完全性を保護してインシデントに関連するデータを保全する.「検査」では，収集したデータからインシデントに関連する情報を評価して抽出する．ログの絞り込みや削除されたファイルの復元，マルウェア挙動の特定などがこの工程に含まれる.「分析」では，関連する情報を抽出した後に複数のソースのデータを関連付けて結論を導き出す.「報告」では，判明した事象や確定まで至らない事象, 実行された措置や行うべき対策，その他の改善に関する勧告など，得られた結論を整理して報告する.

インシデント対応の初動においては，特に被害が拡大しないよう暫定措置をとる対応が最優先で求められる．そのため，ネットワークや端末のログから確実に被害範囲を特定することが求められることがある．フォレンジックにおいてマルウェア解析は「検査」の工程で行われる．一般的

†1 情報セキュリティ大学院大学
Institute of Information Security
†2 NTT コムセキュリティ株式会社
NTT Com Security (Japan) KK

なマルウェア解析の工程は図 1 に示すとおりである[5]．この中で，マルウェアの解析手法は動的解析と静的解析の 2 種類に分類されている．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．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や，RAT（Remote Administration Tool）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．そのため，マルウェアの持つ全ての挙動を解明するのではなく，特定する目標を定めて部分的にアセンブリの解析を行う．

1. マルウェア検体の入手
2. 解析専用環境の設置
3. 動的解析
4. パックされたマルウェア検体の解凍
5. 静的解析
6. マルウェア検体の特徴を特定

図 1 マルウェア解析の工程
Figure 1 Malware analysis process.

インシデントの初動対応において特に必要になる情報がある．例えばマルウェアの接続先の URL が判明すると，ネットワークログから同じマルウェアに感染した端末が特定できる. また, 暗号アルゴリズムや暗号鍵が判明すると，ネットワーク通信のデータから送受信されたデータが特定できる．実際のインシデントでは，複数のインシデントにおいてマルウェアの亜種が共通的に使用されることがあるが，以前解析したことのあるマルウェアと同じ，もしくは類似する挙動を持つマルウェアを解析する場合に，以前解析した関数に相当するコードが判明すると，マルウェア解析にかける時間が削減できる．

本研究では，インシデント対応におけるマルウェアの静的解析を効率化するため，過去に調査したことのあるマルウェアの関数を入力として，制御フローグラフの編集距離を利用することで，解析するマルウェアにおける該当の関数を検索するアルゴリズムを提案する．正常プログラムによる評価では GNU bash と binutils の 11049 個の関数の 90.3% について，マルウェアによる評価では 20 個の関数の 95.0% について，上位 10 位以内に正しく正解を出力できることを示した．

2. 関連研究

本章では，インシデント対応を目的としたマルウェア解析において，マルウェアの特定の挙動を解析するための関連研究について示す．

2.1 暗号アルゴリズムと実行コードの特定

プログラムのソースコードが入手できない場合に，逆アセンブルされた実行命令列からリバースエンジニアリングでプログラムの仕様を調査する．その中でも，プログラムの中で使用された暗号アルゴリズムと暗号処理に関わる命令列の箇所を特定する研究が行われている．

Gröbert らは, プログラムを動作させて実行命令列を取得し，複数の経験則を組み合わせることで暗号アルゴリズムを含む関数の場所を特定し，平文，暗号文，暗号鍵の組み合わせを取得して既存アルゴリズムと比較することで暗号アルゴリズムを特定する手法を提案している[6]．Calvet らは，難読化されたプログラムにおいても暗号アルゴリズムの特定を可能とするため，命令単位の実装に依存しない入出力パラメーターの特定手法を提案している[7].

2.2 プログラム間における関数の対応と差分の特定

リバースエンジニアリングにおいてソフトウェアのセキュリティパッチによる変更内容を解析するため，2 つの実行ファイルを入力として，プログラム間で対応する箇所や差分を特定する研究が行われている．

バイナリファイルにおけるバイト単位の値を単純に比較し，一致や差分を特定する実装がある[8][9][10]. ただし，コンパイラによって命令順序が入れ替わる，異なるレジスタが割り当てられる，異なる命令を使用するなど，同一のソースコードであってもコンパイラや最適化オプションによって出力される実行コードは多様である．これらを全て差分と判断してしまうため，パッチ解析を目的とするリバースエンジニアリングにおいては実用的でない．

そこで, 2 つの逆アセンブルされた命令列を入力として，動作上の一致もしくは差分を特定する研究が行われている．Flake は, 逆アセンブリを関数単位で分割して有向グラフで表現したコールグラフと，さらに関数を制御命令単位で分割して有向グラフで表現した制御フローグラフを定義し，コールグラフのマッチング問題を解く経験則的なアルゴリズムを構築することで，2 つのプログラムの一致と差分を特定する手法を提案している[11]．Dullien らはこの手法を拡張し，Property 関数を用いて比較範囲を適切に限定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箇所を特定している[12]．また，このアルゴリズムを BinDiff というソフトウェアで実装している[13]．Bourquin らは，BinDiff が対応付けできなかった関数の集合に対して，さらに拡張割り当て問題のアルゴリズムを利用して対応づけを行う BinSlayer という手法を提案している[14]．Gao らは，最大共通部分グラフ（MCS）を取得する近似アルゴリズムを用いて，コー

ルグラフと制御フローグラフそれぞれにおいて，一致強度をパラメーターに関数やベーシックブロックの対応を発見する BinHunt という手法を提案し, gzip と tar のパッチによる差分を抽出している[15]．Ming らは，BinHunt において関数境界が正しく取得できない状況を想定し，関数単位ではなくベーシックブロック単位で比較する iBinHunt という手法を提案し，thttpd と gzip におけるパッチの修正箇所を特定している[16]．その他にも，グラフを利用してプログラムの対応と差分を出力する実装が存在する[17][18][19][20].

2.3 再利用されたコードの特定

コードの盗用や使いまわしの特定，フォレンジック作業の効率化のため，コンパイルされたプログラムを対象として再利用されたコードを特定する研究が行われている．

LeDoux らは，関数に含まれる実行コードを抽象化した上でそれぞれの関数を複数のハッシュ値で表現して関数を特定する FuncTracker という手法を提案している[21]．ただし，ハッシュ値を利用するため，命令や命令順序の変化に対しては同一の関数と見なせないという課題があった．Ruttenberg らは，2 回のクラスタリングを行うことで，関数単位ではなくソフトウェアコンポーネント単位の再利用を特定する手法を提案している[22].

また，コンパイルされたプログラムから，コードの使い回しによって伝搬的に発生したコードクローン脆弱性を特定する研究が行われている．Pewny らは，過去に発見された脆弱性の機械語命令列を正規化して式木として表現してこれをシグネチャとし，検査対象の命令列から生成された式木との編集距離を算出することで脆弱性を発見する手法を提案している[23]．中島らは，機械語命令列の正規化を行った上で，局所的な類似度を算出する独自の文字列検索アルゴリズムを適用しコードクローン脆弱性を発見する手法を提案している[24].

2.4 マルウェアの分類

プログラムにおいて関数やコードを特定する手法を応用し，マルウェアを分類する研究が行われている．大量のマルウェアを比較する必要があるため，プログラムの特徴を抽象化して計算を高速化している．

機械語命令列の特徴を利用してマルウェアを分類する手法が提案されている．岩村らは，逆アセンブリにおける機械語命令列の最長共通部分列（LCS）の長さを用いてマルウェアの類似度を定義する手法を提案している[25]．Karim らは n-gram を拡張した n-perm という機械語命令列の順序の変化に強い指標を利用した分類手法を提案している[26]．Gheorghescu は，ベーシックブロック単位で命令列の編集距離を算出して，これを類似度として分類する手法を提案している[27].

グラフを利用してマルウェアを分類する手法も提案されている．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 推移グラフを構築してマルウェアを分類する手法を提案している[28]．Hu らは，2 つのコールグラフ間の類似度を求めるための近似アルゴリズムと，複数のマルウェアと比較するためのインデックス構造を提案し，大量のマルウェアを分類する SMIT というマルウェア管理システムを構築している[29]. Kinable らは，コールグラフ間の編集距離を求める近似アルゴリズムを用いてマルウェアを比較し，k-medoid と DBSCAN というクラスタリング手法を用いて評価を行っている[30].

2.5 関連研究における課題

インシデント対応を目的としたマルウェア解析を効率化するため，解析したい実行コードを特定する関連研究について示した．2.1 節の暗号アルゴリズムと実行コードを特定する手法は，単体のマルウェアを用いて適用できる一方で，暗号のように特定の処理だけを対象としたものであった．2.2 節のプログラム間における関数の対応と差分を特定する手法は，リバースエンジニアリングの中でも主にパッチの修正箇所の特定を目的としており，2 つのプログラム間における関数の対応と差分を特定する手法であった．2 つのプログラムにおける関数を 1 対 1 で対応するものであり，誤って対応付けされた関数に関してはそれ以上の情報が得られなかった．2.3 節の再利用されたコードを特定する手法は，一般的な開発者によって開発されたプログラムを想定していた．コンパイラやコンパイルオプション等の違いによる実行コードの変化を考慮しているが，マルウェアのように意図的に変換されたプログラムには適していなかった．2.4 節のマルウェアの分類は，マルウェアの類似度を求める手法であり実行コードの特定は行っていなかった．

3. BinDiff のマルウェア解析への活用

複数のインシデントにおいてマルウェアの亜種が共通的に使用されることがある．以前解析したことのあるマルウェアと同じ，もしくは類似する挙動を持つマルウェアであった場合に，以前解析した関数に相当する部分が分かると, マルウェア解析にかける時間が削減できる. 本章では，BinDiff の要件とアルゴリズムについて示し，BinDiff を活用して特定の処理を検出することを考察する．

BinDiff は逆アセンブラの IDA Pro[36]が出力した逆アセンブリ，コールグラフ，制御フローグラフなどのデータを入力として用いる．コールグラフは関数の呼出関係を示す有向グラフである．コールグラフのノードはさらに制御フローグラフと呼ばれ，jmp 系命令による分岐で命令列をベ

ーシックブロック単位で分割し，これをノードとした有向グラフとなる．高レベル言語で記述された構造化プログラムは，一般にこのような有向グラフの入れ子構造で表現される．

次に，BinDiff のアルゴリズムについて説明する．BinDiff はコールグラフのノードとして示される各関数に対して（ベーシックブロックの数，ベーシックブロック間の辺の数，関数呼び出しの数）という 3 次元の特徴量を定義し，経験則的なアルゴリズムを用いてこのグラフのマッチング問題を解く．

アルゴリズム 1 に示す initialMatches では，初期状態の構築を行う．ここでは各関数に対して Selector 関数 ε を適用し，一致すると判断できる関数の対応付けを行う．Selector は 2 つの関数の間で特徴量がユニークなものを選択する関数である．

アルゴリズム 2 に示す propagateMatches では，この初期状態で構築した対応付けを拡大する．具体的には，すでに対応づけられた関数に対して Property 関数 π で定まる親子関係に着目し，この範囲に限定して対応付けを行う．これを，アルゴリズム 3 で示す binDiff において，全ての関数に対して評価できるまで繰り返す．また，対応づけが行われなかった関数を差分として出力する．

アルゴリズム 1　initialMatches
Algorithm 1　initialMatches

```
1  function initialMatches(S_A, S_B)
2  M ← ∅
3  foreach vertex a_i ∈ S_A do
4    foreach Selector ε do
5      if (a_i, b_j) ← ε(a_i, S_B) then
6        M ← M ∪ {a_i ↦ b_j}
7        S_A ← S_A ¥ {a_i}
8        S_B ← S_B ¥ {b_j}
9        break
10 return (M, S_A, S_B)
```

アルゴリズム 2　propagateMatches
Algorithm 2　propagateMatches

```
1  function propagateMatches(M, S_A, S_B)
2  foreach {a_i ↦ b_j} ∈ M do
3    foreach Property π do
4      S'_A ← π(a_i, S_A);
5      S'_B ← π(b_j, S_B);
6      if S'_A≠∅ ∧ S'_B≠∅ then
7        foreach vertex a'_i ∈ S'_A do
8          foreach Selector ε do
9            (a'_i, a'_j) ← ε(a'_i, S'_B) then
10             M ← M ∪ {a'_i ↦ b'_j}
11             S'_A ← S'_A{a'_i}
12             S'_B ← S'_B{b'_j}
13             S_A ← S_A ¥ {a'_i}
14             S_B ← S_B ¥ {b'_j}
15           break
16 return (M, S_A, S_B)
```

アルゴリズム 3　binDiff
Algorithm 3　binDiff

```
1  function binDiff(G_A, G_B)
2  S_A ← G_A
3  S_B ← G_B
4  M' ← ∅
5  (M, S_A, S_B) ← initialMatches(S_A, S_B)
6  while M' ≠ M do
7    M' ← M
8    (M,S_A,S_B) ← propagateMatches(M,S_A,S_B)
9  return (M, S_A, S_B)
```

BinDiff はプログラム全体に含まれる関数群を高い精度で対応付けする反面，途中の対応付けが正しいと仮定しながら処理を行う貪欲アルゴリズムであるため，途中で判定を誤ると連鎖的に間違えるという課題がある．また，1 つの関数に対し 1 つしか対応する関数の候補を出力しないため，誤った出力に対しては何も情報が残らない．特に，マルウェアのような解析を妨害する機能が実装されたプログラムにおいては，十分な精度が得られないことが分かっている．

4. 提案方式

4.1 要件

インシデント対応におけるマルウェアの静的解析では特定の処理に着目して解析を行う．BinDiff のように 2 つのコールグラフ全体を入力として全ての関数の対応付けを求める必要はないが，逆に解析したい特定の関数についてはより高い精度での対応付けが求められる．ここで，対応付けに自信がない場合であっても複数の候補を出力することができれば，マルウェア解析者はその中から正解を探して速やかに解析にとりかかることができる．検索したい 1 つの関数と，検索される関数全体を入力として，一致すると考えられる関数を「検索」して候補を複数出力する方式が考えられる．

マルウェアの静的解析は一般的に市販されている PC を使用する．本方式も静的解析を行う PC の性能で動作することを想定する．また，マルウェアに含まれる全ての関数を対応づける必要がないため，BinDiff のように貪欲アルゴリズムを用いて実行時間を最小化する必要はなく，探索アルゴリズムを用いてより高い精度の結果を追及することができる．

本方式では BinDiff を代表とする従来研究と同様に，逆アセンブルにより関数構造が正しく復元できていることを前提とする．また，プログラムによっては関数名がシンボルとして含まれており関数の比較に利用できることがある

が，マルウェアの場合は通常シンボルは可能な限り削除されるため，外部関数名など最低限のシンボルしか利用できないという状況を想定する．

4.2 定義

本アルゴリズムでは，関数が一致するかどうか判断するために，グラフにおける編集距離を利用する．2 つのグラフの編集距離は，ノードとエッジの挿入，削除，置換で構成される編集操作を使用して，一方のグラフをもう一方のグラフに変換するのに必要となる最小の編集数で定義される．グラフがループ構造をもたない場合，これをツリーと呼ぶ. 特に, ツリーが親子関係を持つ場合は順序木と呼び，順序木のノードがラベルを持つ場合はラベル付き順序木と呼ぶ.

4.3 グラフの比較と編集距離

プログラムの比較研究において，しばしばグラフの一致問題を解決する必要がある．2 つのグラフに対して共通する最大の部分グラフを求める最大共通部分グラフ（MCS）問題は NP-Hard であることが知られている[31].

また，グラフの類似性を定義する指標として，2 つのグラフの共通部分グラフを求める代わりに，グラフ間の編集距離を利用することができる．編集距離は，ノードの追加，削除とラベルの変更によってグラフを編集する場合のグラフ間の最小の編集数で定義される．一般的なグラフに対して編集距離を求める問題は NP-Hard であることが知られている．パターン認識の分野では大きいグラフに対して計算する必要があるため，効率的に近似解を求めるアルゴリズムが研究されている[32].

ラベル付き順序木に対する編集距離としては，Zhang の計算量 n^4 のアルゴリズム[33]と Klein の計算量 $n^3 \log n$ のアルゴリズム[34]が知られている．ただし，最適なアルゴリズムは入力するグラフに依存する[35].

4.4 提案アルゴリズム

提案方式で使用するアルゴリズムをアルゴリズム 4，5 に示す．`generateCFTree` はコールグラフ `G` と記憶変数 `m` を入力として，制御フローグラフに近似するコントロールフローツリーを出力する関数である．コールグラフ `G` を深さ優先で探索し，`T` のノードとして追加するが，一度出現したベーシックブロックを発見した場合はそれ以降の探索を行わない．さらに，各ベーシックブロックにおける call 命令が呼び出す外部関数名に着目し，同じ関数を呼び出すブロックに対しては関数名をラベルとして割り当てる．複数の関数を呼び出すベーシックブロックに対しては複数の関数名を辞書順にソートして結合した文字列をラベルとする．また，call 命令を含まないベーシックブロックについては，すでに割り当てた記号とは異なる共通のラベルを割り当てる．

次に，`binGrep` はグラフ G_A に出現する関数 f_A とグラフ G_B を入力として，関数 f_A と G_B に含まれる全ての関数について，元の制御フローグラフに近似する制御フローツリーを生成し，ツリーに対する編集距離を計算する．最後に距離の短いものから順番にソートして出力する．

アルゴリズム 4　generateCFTree
Algorithm 4　generateCFTree

```
1  function generateCFTree(G, m)
2  T ← Node(G)
3  for H ← Child(G) do
4    if H ∉ m do
5      m.append(H)
6      T.addChild(generateCFTree(H, m))
7  return T
```

アルゴリズム 5　binGrep
Algorithm 5　binGrep

```
1  Function binGrep(f_A, G_B)
2  r ← ∅
3  t_A ← generateCFTree(f_A, ∅)
4  for x ← G_B do
5    u ← generateCFTree(x, ∅)
6    r.append(treeEditDistance(t_A, u))
7  a ← sort(r)
8  return a
```

5. 評価

本アルゴリズムの有効性について，正規のプログラムとマルウェア検体を対象に評価を行った.

BinDiff は，逆アセンブラ IDA Pro が出力した逆アセンブリとコールグラフ，制御フロー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と差分を出力する．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.

提案方式は IDA Pro のプラグインとして実装し，IDA Pro が出力した逆アセンブリとコールグラフ，制御フローグラフを使用して計算を行う．また，ラベル付き順序木の編集距離の計算は，Zhang のアルゴリズムを実装した Python ライブラリ[37]を利用した.

マルウェア解析者は一般的な PC を使用することを想定した．評価に用いた PC の仕様は，CPU Intel Core M-5Y71 1.20GHz，メモリー容量 8GB，SSD 256GB，Windows 8 64bit である.

5.1 GNU bash による評価

GNU bash を用いて提案方式の評価を行った．使用したバージョンは表 1 のとおりである.

表 1 比較に使用した GNU bash のバージョン

Table 1 GNU bash version

No.	評価プログラム	公開時期
1	bash 4.0	2009 年 2 月 20 日
2	bash 4.3.30	2014 年 11 月 7 日

2 つのバージョンアップ前後のプログラムにおいて，同じ名前を持つ関数を対応する関数として正解を定義する．評価ではこの 2 つのプログラムに対して strip コマンドでシンボルを削除し，シンボルが削除された関数に対して，どれだけ正解に近づくことができるか評価を行った．実際に解析者が利用する際に検索結果を確認できる現実的な範囲として，ここでは 10 位未満を正解と定義した．

GNU bash 4.0 においてシンボルが削除された関数 381 個について評価を行った結果を図 2 に示す．314 個の関数については，対応する関数が正しく検索結果の 1 位として出力された．また，38 個の関数については検索結果のランク外となった．

また，BinDiff の出力と比較した結果を表 2 に示す．BinDiff は 345 個の関数について正しく正解を出力したことに対し，提案方式では 343 個の関数を正しく検索することができ，ほぼ同精度の結果を達成することができた．特に，BinDiff が不正解を出力した 36 個の関数のうち，29 個（80.6%）の関数について正しい結果を示すことができたため，BinDiff と組み合わせることで，より精度の高い対応付けが実現できたと考えられる．

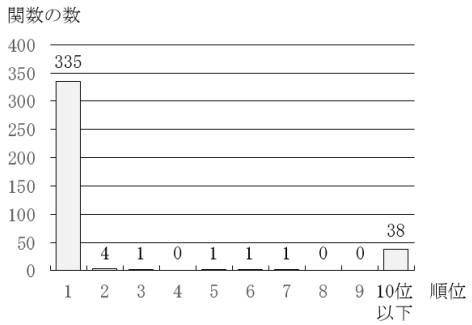

図 2 GNU bash における検索順位

Figure 2 Rank result of GNU bash.

表 2 提案方式による GNU bash の評価

Table 2 Experiment result of GNU bash comparison.

		BinDiff		合計
		正解	不正解	
提案方式	正解	314	29	343
	不正解	31	7	38
合計		345	36	381

実行時間については，BinDiff はコールグラフ全体に対して 3.7 秒であったが，提案方式では 1 つの関数の検索に対して平均 11.6 秒，最大実行時間は 437 秒であった．

5.2 GNU binutils による評価

GNU binutils を用いて同様に提案方式の評価を行った．binutils には 15 個のプログラム addr2line, ar, as, c++filt, elfedit, gprof, ld, nm, objcopy, objdump, ranlib, readelf, size, strings, strip が含まれる．使用したバージョンは表 3 のとおりである．

表 3 比較に使用した GNU binutils のバージョン

Table 3 GNU binutils versions.

No.	評価プログラム	公開時期
1	binutils 2.22	2011 年 11 月 21 日
2	binutils 2.25.1	2015 年 7 月 21 日

GNU binutils 2.22 においてシンボルが削除された関数 10668 個について，BinDiff の出力と比較した結果を表 4 に示す．BinDiff は 9635 個の関数について正しく正解を出力したことに対し，提案方式では 9651 個の関数を正しく検索することができ，ほぼ同精度の結果を達成することができた．特に，BinDiff が不正解を出力した 1033 個の関数のうち，674 個（65.2%）の関数について正しい結果を示すことができた．

表 4 提案方式による GNU binutils の評価

Table 4 Experiment result of GNU binutils comparison.

		BinDiff		合計
		正解	不正解	
提案方式	正解	8977	674	9651
	不正解	658	359	1017
合計		9635	1033	10668

実行時間については，BinDiff はコールグラフ全体に対して 4.5 秒であったが，提案方式では 1 つの関数の検索に対して平均 11.7 秒，最大実行時間は 615 秒であった．

5.3 マルウェアによる評価

次に，マルウェア検体を用いて評価を行った．表 5 は一連のインシデントで用いられたマルウェアの亜種であることが分かっており，使用された時期によってバージョンが異なる．

表 5 マルウェア検体

Table 5 Malware Samples.

No.	バージョン	評価における役割
検体 1	t17.08.21	既知マルウェア
検体 2	t17.08.26	解析対象マルウェア
検体 3	t17.08.30	解析対象マルウェア
検体 4	t17.08.30	解析対象マルウェア
検体 5	t17.08.31	解析対象マルウェア

最も古いバージョンを持つマルウェア検体 1 を解析したことがあるマルウェアとし，検体 2〜5 をその後のインシデ

ントで解析する必要マルウェアと想定した.

それぞれの検体に対して, BinDiff と提案方式で比較を行った結果を表 6～表 9 に示す．バージョンの近い検体 1 と検体 2，検体 1 と検体 3 の間では，BinDiff と提案方式ともに全て正解を出力した．検体 1 と検体 4 の間では BinDiff は sub_40CA5E が対応づけを誤って出力したが，提案方式では正解を出力することができた．検体 1 と検体 5 の間では，BinDiff は 4 つの関数 sub_401B19，sub_401AC1，sub_40801C，sub_40CA5E について対応づけを誤ったが，そのうち 3 つに対して提案方式で正解を検索することができた.

表 6 検体 1 vs 検体 2 の比較結果

Table 6 Comparison result between malware sample 1 and 2.

	検体 1	検体 2	BinDiff	提案方式
関数 1	sub_401B19	sub_401B19	○	○ 1st
関数 2	sub_401AC1	sub_401AC1	○	○ 1st
関数 3	sub_40801C	sub_40860E	○	○ 1st
関数 4	sub_40CA5E	sub_40CFE1	○	○ 1st
関数 5	sub_40204D	sub_40204D	○	○ 1st

表 7 検体 1 vs 検体 3 の比較結果

Table 7 Comparison result between malware sample 1 and 3.

	検体 1	検体 3	BinDiff	提案方式
関数 1	sub_401B19	sub_401B1F	○	○ 1st
関数 2	sub_401AC1	sub_401AC6	○	○ 1st
関数 3	sub_40801C	sub_40816E	○	○ 1st
関数 4	sub_40CA5E	sub_40CB28	○	○ 1st
関数 5	sub_40204D	sub_401DA2	○	○ 1st

表 8 検体 1 vs 検体 4 の比較結果

Table 8 Comparison result between malware sample 1 and 4.

	検体 1	検体 4	BinDiff	提案方式
関数 1	sub_401B19	sub_401B1F	○	○ 1st
関数 2	sub_401AC1	sub_401AC6	○	○ 1st
関数 3	sub_40801C	sub_408A02	○	○ 3rd
関数 4	sub_40CA5E	sub_40D171	×	○ 5th
関数 5	sub_40204D	sub_401D98	○	○ 1st

表 9 検体 1 vs 検体 5 の比較結果

Table 9 Comparison result between malware sample 1 and 5.

	検体 1	検体 5	BinDiff	提案方式
関数 1	sub_401B19	sub_403290	×	× 164th
関数 2	sub_401AC1	sub_403210	×	○ 1st
関数 3	sub_40801C	sub_40D840	×	○ 2nd
関数 4	sub_40CA5E	sub_4156F0	×	○ 1st
関数 5	sub_40204D	sub_403640	○	○ 1st

6. まとめと今後の課題

近年, 日本においても広範囲な APT 攻撃による大規模な被害を経験し, インシデント対応の重要性が再認識された. インシデント対応の初動において迅速に被害範囲を特定するため，マルウェア解析で特定の挙動の調査を行うことがある．複数のインシデントにおいてマルウェアの亜種が共通的に使用されることがあるが，以前解析したマルウェアの関数に相当するコードの場所を特定できると，マルウェア解析にかける時間が削減できる．このような場合に，過去に解析したマルウェアの情報を用いて，制御フローグラフの編集距離を用いた関数の比較を行うことで特定の関数を検索するアルゴリズムを提案した.

正常プログラムによる評価では GNU bash と binutils の 11049 個の関数の 90.3% について，マルウェアによる評価では 20 個の関数の 95.0% について，現実的な時間内で上位 10 位以内に正しく正解を出力できることを示した.

今後の課題としては，グラフの特徴だけでなく機械語命令列の特徴を利用してより精度の高いアルゴリズムを構築することが挙げられる．特に，制御フローグラフが小さい関数が数多く存在した場合に特徴が捉えづらく誤った判定をしてしまうため，機械語命令列の特徴を加味することが必要となる．また，本方式は検索したい関数が存在しない場合でも関数の候補を上位から出力していたが，このような場合は該当する関数が存在しないことを示すアルゴリズムが求められる．アルゴリズムを改善してグラフの比較計算を高速化することも必要である.

参考文献

[1] 株式会社カスペルスキー: BLUE TERMITE ～日本を標的にする APT 攻撃～（オンライン），入手先〈http://media.kaspersky.com/jp/pdf/pr/Kaspersky_BlueTermiteDaily-PR-1016.pdf〉（参照 2016-03-06）.

[2] 朝長秀誠，中村祐 : CODE BLUE 2015 日本の組織をターゲットにした攻撃キャンペーンの詳細，JPCERT/CC（オンライン），入手先〈https://www.jpcert.or.jp/present/2015/20151028_codeblue_ja.pdf〉（参照 2016-03-06）.

[3] 船越絢香，中村祐，竹田春樹 : 標的型攻撃で用いられたマルウェアの特徴と攻撃の影響範囲の関係に関する考察，コンピュータセキュリティシンポジウム 2015 論文集 (CSS2015), vol.2015, No.3, pp.963-970 (2015).

[4] Kent, K., Chevalier, S., Grance, T. and Dang, H.: Guide to Integrating Forensic Techniques into Incident Response, National Institute of Standards and Technology (online), available from 〈http://csrc.nist.gov/publications/nistpubs/800-86/SP800-86.pdf〉 (accessed 2016-03-06).

[5] 新井悠，岩村誠，川古谷裕平，青木一史，星澤裕二 : アナライジング・マルウェア フリーツールを使った感染事案対処，オライリー・ジャパン (2010).

[6] Gröbert, F., Willems, C. and Holz, T.: Automated identification of cryptographic primitives in binary programs, *Proc. 14th International Symposium Recent Advances in Intrusion Detection* (*RAID 2011*), pp.41-60, Springer (2011).

[7] Calvet, J., Fernandez, J. M. and Marion, J.: Aligot: Cryptographic Function Identification in Obfuscated Binary Programs, *Proc.*

ACM Conference on Computer and Communications Security (*CCS 2012*), pp.169-182, ACM (2012).

[8] Percival, C.: Naive differences of executable code, Binary diff/patch utility (online), available from〈http://www.daemonology.net/papers/bsdiff.pdf〉(accessed 2016-03-06).

[9] MacDonald, J.: Open-source binary diff differential compression tools VCDIFF (RFC 3284) delta compression, xdelta (online), available from〈http://xdelta.org〉(accessed 2016-03-06).

[10] Heirbaut, J.: Diff utility for binary files, JojoDiff (online), available from 〈 http://jojodiff.sourceforge.net 〉 (accessed 2016-03-06).

[11] Flake, H.: Structural Comparison of Executable Objects, *Detection of Intrusions and Malware & Vulnerability Assessment* (*DIMVA 2004*), IEEE Computer Society, pp.161–173 (2004).

[12] Dullien, T. and Rolles, R.: Graph-based comparison of Executable Objects, *Proc. of SSTIC 2005* (2005).

[13] Zynamics: Zynamics BinDiff (online), available from〈http://www.zynamics.com/bindiff.html〉(accessed 2016-03-06).

[14] Bourquin, M., King, A. and Robbins, E.: BinSlayer: Accurate Comparison of Binary Executables, *Proc. 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop* (*PPREW 2013*), ACM (2013).

[15] Gao, D., Reiter, M. K. and Song, D.: Binhunt: Automatically finding semantic differences in binary programs, *Proc. 10th International Conference on Information and Communications Security* (*ICICS 2008*), pp.238-255, Springer (2008).

[16] Ming, J., Pan, M. and Gao, D.: iBinHunt: Binary Hunting with Inter-procedural Control Flow, *Proc. Information Security and Cryptology* (*ICISC 2012*), pp.92-109, Springer (2012).

[17] Oh, J.: Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries, *Proc. Black Hat USA 2009* (2009).

[18] Tenable Network Security: PachDiff2 High Performance Patch Analysis (online), available from〈https://www.tenable.com/blog/patchdiff2-high-performance-patch-analysis 〉(accessed 2016-03-06).

[19] eEye Digital Security: eEye Binary Diffing Suite (online), available from〈https://web.archive.org/web/20080705014733/http://research.eeye.com/html/tools/RT20060801-1.html 〉(accessed 2016-03-06).

[20] Zimmer, D.: IDACompare, VeriSign iDefense Labs (online), available from〈http://sandsprite.com/iDef/IDACompare/〉(accessed 2016-03-06).

[21] LeDoux, C., Lakhotia, A., Miles, C. and Notani, V.: FuncTracker: Discovering Shared Code to Aid Malware Forensics Extended Abstract, *Proc. 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats* (*LEET 2013*) (2013).

[22] Ruttenberg, B., Miles, C., Kellogg, L., Notani, V., Howard, M., LeDoux, C., Lakhotia, A. and Pfeffer, A.: Identifying Shared Software Components to Support Malware Forensics, *Detection of Intrusions and Malware & Vulnerability Assessment* (*DIMVA 2014*), pp.21-40, Springer (2014).

[23] Pewny, J., Schuster, F., Rossow, C., Bernhard, L. and Holz, T.: Leveraging Semantic Signatures for Bug Search in Binary Programs, *Proc. 30th Annual Computer Security Applications Conference Pages* (*ACSAC 2014*), pp.406- 415, ACM (2014).

[24] 中島明日香，岩村誠，矢田健：機械語命令の類似度算出による複製された脆弱性の発見手法の提案，コンピュータセキュリティシンポジウム 2015 論文集 (CSS2015)，vol.2015，No.3，pp.304-309 (2015).

[25] 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，pp.1622-1632 (2010).

[26] Karim, M. E., Walenstein, A., Lakhotia, A. and Parida, L.: Malware phylogeny generation using permutations of code. *Journal of Computer Virology*, Vol.1, Issue 1-2, pp.13-23, Springer-Verlag (2005).

[27] Gheorghescu, M.: An automated virus classification system, *Virus Bulletin Conference 2005* (2005).

[28] 岩本一樹，和崎克己：静的解析により抽出された API 推移に基づくマルウェアの分類，情報処理学会論文誌，Vol.54，No.3，pp.1199-1210 (2013).

[29] Hu, X., Chiueh, T. and Shin, K. G.: Large-Scale Malware Indexing Using Function-Call Graphs, *Proc. 16th ACM conference on Computer and communications security* (*CCS 2009*), pp.611-620, ACM (2009).

[30] Kinable, J. and Kostakis, O.: Malware Classification based on Call Graph Clustering, *Journal in Computer Virology*, Vol.7, Issue 4, pp.233-245, Springer-Verlag (2011).

[31] Levi, G.: A Note on the Derivation of Maximal Common Subgraphs of Two Directed or Undirected Graphs, *Calcolo*, Vol.9, pp.341-354, Springer-Verlag (1973).

[32] Gao, X., Xiao, B., Tao, D. and Li, X.: A survey of graph edit distance, *Pattern Analysis and Applications*, Vol.13, Issue 1, pp.113-129, Springer-Verlag (2010).

[33] Zhang, K. and Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems, *SIAM Journal of Computing*, Vol.18, Issue 6, pp.1245-1262 (1989).

[34] Klein, P.: Computing the edit-distance between unrooted ordered trees, *Proc. 6th Annual European Symposium on Algorithms*, pp.91-102, Springer-Verlag (1998).

[35] Dulucq, S. and Touzet, H.: Analysis of tree edit distance algorithms, *Proc. 14th annual conference on Combinatorial Pattern Matching* (*CPM 2003*), pp.83-95, Springer (2003).

[36] Hex-Rays: IDA Pro (online), available from〈http://www.hex-rays.com〉(accessed 2016-03-06).

[37] Henderson, T. and Johnson, S.: Zhang-Shasha: Tree edit distance in Python, available from〈https://zhang-shasha.readthedocs.org/en/latest/〉(accessed 2016-03-06).