Computer Security Symposium 2016
11 - 13 October 2016

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 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．マルウェアの亜種が共通的に使用された場合，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速やかに解析が行える．これを実現する代表的な研究成果で，パッチ解析を目的としてプログラムを「比較」する BinDiff というソフトウェアがあるが，貪欲アルゴリズムにより連鎖的に間違えてしまう場合や，間違えた場合に利用できる情報がないという課題があった．本研究では，マルウェア解析を目的として，関数における制御フローグラフの編集距離と命令列を用いて関数を「検索」するアルゴリズムを提案する．GNU bash と binutils では 11049 個の関数のうち 90.3% について正解を出力した．Emdivi の 6 検体の評価ではインシデント対応で鍵となる 15 つの関数の 80% について正解を出力した．また，Emdivi と PlugX のそれぞれ 2 検体の全関数について評価を実施し，マルウェア解析において提案方式が有効であることを示した．

キーワード：フォレンジック，マルウェア解析，静的解析

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 importance of prompt and appropriate incident response is reacknowledged. When resemble malware is used in some attacks commonly, the analyst can proceed static analysis immediately specifying the position of function previously analyzed. There is representative implementation of this functionality named BinDiff. However, there are some problems that BinDiff mistakes continuously because of greedy algorithm, and there are no information against such functions. In this paper, to improve the efficiency of the static analysis in incident response, we propose“searching” algorithm that employs edit distance of the control flow graph and similarity of instructions. We evaluated that 90.3% of the 11049 functions of the GNU bash and binutils as normal program and 80% of the 15 functions of Emdivi as malware can be output correctly. Furthermore, we evaluated all functions of two samples of Emdivi and PlugX, and show proposal method is available for malware analysis.

Keywords: Forensics, Malware Analysis, Static Analysis

1. はじめに

1.1 背景

近年，日本においても APT 攻撃による大規模な被害を経験した．カスペルスキー社は，大規模組織を中心に 300 以上の組織が被害にあったと報告した[1]. 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，脆弱性や入手したパスワードを駆使して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]．この中で，共通的に Emdivi という RAT（Remote Administration Tool）が使用されていた．

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るが，侵入経路や影響範囲を特定するため，しばしばフォレンジックを行う．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おり，マルウェア挙動の特定はこの「検査」工程において実施される．[4]

マルウェア解析手法は主に動的解析と静的解析に分類される[5]．動的解析では，マルウェアを実行して，ファイルやレジストリの操作，ネットワーク通信，API 呼び出しなどを観測する．効率的に解析できる一方，RAT のように攻撃者からの指令で動作するマルウェアを完全に解明する事は難しい．静的解析では，マルウェアの逆アセンブリを取得してコードを読み解く．技術者のスキルと時間を要するため，インシデント対応においては特定の挙動に着目して解析することになる．例えば，プロキシサーバーに記録されている暗号化された C&C 通信を調査するため，接続先のドメインや HTTP リクエストを構築する処理，暗号アルゴリズムや暗号鍵を解析することがある．

インシデントにおいてマルウェアの亜種を解析する場合，以前解析したマルウェアの情報が静的解析に利用できることがある．軽微な変更しか行われていないソースコードからコンパイルされたマルウェアの解析においては，難

†1 情報セキュリティ大学院大学
Institute of Information Security
†2 NTT セキュリティ・ジャパン株式会社
NTT Security (Japan) KK

読化が十分でない場合，予備知識がない状況よりも効率的に解析が行える．そのため，これから解析するマルウェアにおいて，以前解析したマルウェアの関数に相当するコードを特定する技術が求められる．

これを実現する代表的な研究成果として，BinDiff というソフトウェアがある[13]．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 つのプログラムの逆アセンブリを入力として「比較」を行い，相当する関数を対応づけて出力する．BinDiff は逆アセンブリ中に含まれる全ての関数について対応付けを試みるため，プログラムの変更内容を抽出する目的において実力を発揮する．これをマルウェア解析に適用した場合にも一定の成果が得られるが，貪欲アルゴリズムにより探索を行うことで連鎖的に対応付けを間違えてしまう場合があり，さらに間違えた場合は情報が残らないという課題があった．

1.2 本研究における成果

マルウェア静的解析では特定の挙動に着目して解析を行うため，2 つのプログラム全体の変更点を抽出する必要はなく，逆に解析したい特定の関数について高い精度でコードを特定することが求められる．そこで，本研究では制御フローグラフの編集距離と逆アセンブリの一致率を用いた新しいアルゴリズムにより，過去に解析したマルウェアの関数に相当する関数を確度の高い順に「検索」して出力する BinGrep という方式を提案する．マルウェア解析者はこの中から正解を探して速やかに解析作業に取りかかることができる．

提案方式と BinDiff について正常プログラムを用いて評価を行った．提案方式は複数の候補を出力するため上位から 10 位以内を正解と定義した．GNU bash と GNU binutils では，11049 個の関数のうち 90.3% について正解を出力した．特に，BinDiff が失敗した 1069 個の関数のうち 65.8% について提案方式が正解を出力し，BinDiff と提案方式を併用することでより多くの関数を特定できることを示せた．

また，実際に APT 攻撃で使用された RAT である Emdivi と PlugX について評価を行った．Emdivi の 6 個の検体を用いてログ調査のために必要となる HTTP リクエストを構成する処理と暗号処理を行う 3 つの関数を評価した結果，80%について正解を出力した．さらに，Emdivi と PlugX のそれぞれ 2 検体における全関数に対して評価を行い，Emdivi の 1018 個の関数のうち 75%，PlugX の 472 個の関数のうち 85%について正解を出力し，提案方式がマルウェア解析において有効であることを示せた．

2. 関連研究

本章では，インシデント対応におけるマルウェアの静的解析において，マルウェアのコードを特定することを目的とした関連研究を示す．

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る．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る．2 つのプログラムにおける関数を 1 対 1 で対応するが，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い．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る．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い．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い．

3. BinDiff のマルウェア解析への活用

マルウェアの亜種を解析するにあたって，以前解析した処理に相当するコードが特定できると，速やかに作業に取りかかれる．2.2 節で取り上げた BinDiff を利用すると 2 つのプログラム間における関数の対応付けを取得できる．本章では BinDiff のアルゴリズムを示し，マルウェア解析における課題について説明する．

BinDiff は逆アセンブラの IDA Pro[36]が出力した逆アセンブリを受け取る．高レベル言語で開発された構造化プログラムは，一般的にコールグラフと制御フローグラフというグラフの入れ子で表現することができる．コールグラフはプログラムに含まれる関数を頂点，関数の呼び出しを辺で定義した有向グラフである．1 つの関数に含まれる逆アセンブリは，制御転送命令（jmp 系命令）が現れる箇所で分割され，これを基本ブロックと呼ぶ．制御フローグラフは基本ブロックを頂点，制御転送命令によるジャンプを辺で定義した有向グラフである．また，制御フローグラフ，すなわちコールグラフの各頂点に対して（基本ブロックの数，基本ブロック間の辺の数，関数呼び出しの数）という 3 次元の特徴量を与える．

BinDiff は 2 つのプログラムの逆アセンブリを入力としてそれぞれコールグラフと 3 次元の特徴量を構築する．そ

してアルゴリズム 3 の binDiff の中でアルゴリズム 1 の initialMatches と，アルゴリズム 2 の propagateMatches を適用して処理を行う．

アルゴリズム 1　initialMatches
Algorithm 1　initialMatches

```
1  Function initialMatches(SA, SB)
2    M ← ∅
3    foreach vertex ai ∈ SA do
4      foreach Selector ε do
5        if (ai, bj) ← ε(ai, SB) then
6          M ← M ∪ {ai ↦ bj}
7          SA ← SA \ {ai}
8          SB ← SB \ {bj}
9          break
10   return (M, SA, SB)
```

アルゴリズム 2　propagateMatches
Algorithm 2　propagateMatches

```
1  Function propagateMatches(M, SA, SB)
2    foreach {ai ↦ bj} ∈ M do
3      foreach Property π do
4        S′A ← π(ai, SA);
5        S′B ← π(bj, SB);
6        if S′A ≠ ∅ ∧ S′B ≠ ∅ then
7          foreach vertex a′i ∈ S′A do
8            foreach Selector ε do
9              if (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               SA ← SA \ {a′i}
14               SB ← SB \ {b′j}
15               break
16   return (M, SA, SB)
```

アルゴリズム 3　binDiff
Algorithm 3　binDiff

```
1  Function binDiff(GA, GB)
2    SA ← GA
3    SB ← GB
4    M′ ← ∅
5    (M, SA, SB) ← initialMatches(SA, SB)
6    while M′ ≠ M do
7      M′ ← M
8      (M, SA, SB) ← propagateMatches(M, SA, SB)
9    return (M, SA, SB)
```

initialMatches では最初のマッチングを行う．Selector 関数 ε は 2 つのプログラム間の頂点における特徴量がユニークに一致するものを探し，これを最初の頂点の対応付けとする．propagateMatches は，2 つのプログラム間での頂点の対応付けの範囲を拡大する．Property 関数 π は頂点に対する隣の頂点の集合を返す関数であり，この範囲で再び Selector 関数 ε を適用して特徴量がユニークに一致するものをさがす．binDiff では initialMatches を適用し，全ての関数が評価できるまで propagateMatches を繰り返す．

BinDiff は，2 つのプログラムにおいて同じ関数から呼び出された関数は同一である可能性が高いという前提に基づき，関数の呼び出し関係に着目しながら関数の対応付けを行う．逆アセンブリ中に含まれる全ての関数について対応付けを試みるため，プログラムの変更内容を抽出する目的において実力を発揮する．BinDiff をマルウェア解析に適用した場合にも一定の成果が得られるが，途中の対応付けが正しいと仮定して関数の呼び出し関係を辿りながら処理を続ける貪欲アルゴリズムであるため連鎖的に判断を誤りやすい．さらに，それぞれの関数に対して対応する関数を 1 つしか出力しないため，出力が誤っていた場合は何も情報が残らないという課題がある．これらの理由により，マルウェアのように意図的に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，BinDiff では目的のコードを検出する上で十分な情報が得られない場合がある．

4.　提案方式

4.1　要件

本節では，提案方式に求められる要件について示す．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挙動に着目して解析を行う．まずは過去に解析したマルウェアとこれから解析するマルウェア亜種を BinDiff に適用して全ての関数について対応付けを行い，解析したい関数に相当するコードを探す．BinDiff の出力を確認して正解であると判断すれば解析者はそのままアセンブリを読み解く．誤った出力であると判断した場合は，提案方式に対して，過去に解析したマルウェアの逆アセンブリ，解析したい処理を行う関数，マルウェア亜種の逆アセンブリを入力として，マルウェア亜種において相当する可能性のあるコードを「検索」する．

提案方式では，相当する可能性のある関数の候補を確度の高い順に複数出力することで，1 位の出力が誤っていたとしてもマルウェア解析者が下位の出力から正解を探して解析に取りかかれる必要がある．この際，複数の検索結果に対してどれが求めるコードであるかマルウェア解析者が速やかに判断できるよう，逆アセンブリを並べて比較できるインターフェースとなることも考慮する．

マルウェアの静的解析は市販の PC を使用することが一般的であり，解析作業に支障のない待ち時間で処理が完了することを想定する．

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．プログラムによっては関数名がシンボルとして含まれ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が削除されるため，外部関数名など最低限のシンボルしか利用できない状況を想定する．

4.2　定義

本アルゴリズムでは，関数が似ているか判断するために有向グラフの編集距離を利用する．頂点にラベルが付与されたグラフにおける編集操作を，頂点と辺の挿入，頂点と辺の削除，頂点のラベルの置換という操作で定義する．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操作の数を有向グラフの編集距離と定義する．

有向グラフの中でも，根と呼ばれる頂点が存在し，連結で閉路がないものを木と呼び，頂点がラベルを持つ場合はラベル付き木と定義する．

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．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

本節では提案方式のアルゴリズムについて説明する．4.1 節の要件を満たすため，関数間の距離を用いることで，相当する可能性の高い複数の関数を順番に出力できるアルゴリズムを構築する．提案方式はアルゴリズム 4 の constructCFT とアルゴリズム 5 の binGrep で構成される．

アルゴリズム 4　constructCFT
Algorithm 4　constructCFT

```
In   G: 制御フローグラフ
     i: 基本ブロック V のインデックス
     m: 訪問済みであることを記録するための記憶領域
Out  G: 制御フローグラフに近似するラベル付き木

1  Function constructCFT(G, i, m)
2    foreach j ∈ JmpTo(G, i) do
3      if j ∈ m do
4        DelJmp(G, i, j)
5      else
6        m ← m ∪ {j}
7        SetLabel(G, j)
8        constructCFT(G, j, m))
```

アルゴリズム 5　binGrep
Algorithm 5　binGrep

```
In   GA: プログラム A における検索元となる関数
     GB*: プログラム B の関数 GB の集合
Out  r': 検索結果のリスト

1  Function binGrep(GA, GB*)
2    r ← ∅
3    SetLabel(GA, 0)
4    constructCFT(GA, 0, ∅)
5    foreach GB ∈ GB* do
6      SetLabel(GB, 0)
7      constructCFT(GB, 0, ∅)
8      r ← r ∪ (EditDistance(GA, GB), LCS(GA, GB))
9    r' ← Sort(r)
10   return r'
```

constructCFT は，グラフ間における編集距離を効率的に計算するため，制御フローグラフ G に対して近似するラベル付き木を出力する関数である．制御フローグラフ G を頂点と辺の集合 G = (V, E) と定義し，頂点を $V = \{V_1, V_2, ..., V_n\}$，辺を $E = \{(V_i, V_j), ...\}$ (i, j は頂点 V のインデックス) と定義する．JmpTo(G, i) は (Vi, *) を満たす辺の集合を返す関数，SetLabel(G, i) は G の頂点 V_i において call 命令で呼び出される外部関数名を連結した文字列を付与する関数，DelJmp(G, i, j) は G の辺から (V_i, V_j) を削除する関数とする．この関数は深さ優先探索により制御フローグラフ G を探索し，記憶領域 m に探索済みの頂点を追加する．ただし，探索済みの頂点を発見した場合は G から辺を削除する．

binGrep は，プログラム A において検索元となる関数 G_A とプログラム B における関数の集合 G_B^*を入力として，G_A と G_B^*の全ての要素と比較し，順位の高い順に出力する関数である．constructCFT を使用してグラフを初期化した上で EditDistance(G_A, G_B) を用いて編集距離を計算し，さらに LCS(G_A, G_B) を用いて命令列の最長共通部分列を計算する．最後に Sort(r) で第 1 要素を昇順に，同着の場合は第 2 要素を降順にソートして出力する．

4.5 実装

IDA Pro が生成した逆アセンブリを含むデータベース（IDB ファイル）を使用し，IDA Pro の API を利用するスクリプトとして提案アルゴリズムを実装した．ラベル付き木における編集距離の計算は，Zhang のアルゴリズムを実装した Python ライブラリ[37]をベースに，Python を C 言語拡張する Cython[38]に移植して高速化したものを使用した．

提案方式の実行例を示す．図 1 は CUI 上で実行した結果である．検索元のプログラムの IDB ファイル，検索元の関数名，検索先のプログラムの IDB ファイルを引数として指定して実行する．結果はデフォルトで 10 位まで出力するが，オプションによって上限なく出力することも可能である．検索結果は編集距離（Dist）が小さい順に並び，さらに同着の関数は命令の一致率（Sim）が大きい順に並ぶ．最小の編集距離を持つ関数名の右には「*」印が付与される．

```
$ bingrep.py Emdivi_01.idb sub_405954 Emdivi_02.idb
Src Program  is ‘Emdivi_01’
Src Function is ‘sub_405954’
Dest Program is ‘Emdivi_02’
-------------------------------------------------
No. Dist  Sim:   Dst Function
  1    0  100:     sub_405D1B *
  2    0   99:     sub_410BE1 *
  3    4   70:     sub_40BF28
  4   13   16:       __alldvrm
  5   15   15:        __alldiv
  6   15   13:        __allrem
  7   15   12:         _memset
  8   15    8:         _strcmp
  9   16   19:     sub_40DB78
  9   16   19:     sub_4035BF
```

図 1　提案方式の実行例（CUI）
Figure 1　Execution example of proposal method (CUI).

マルウェア解析者が正解を確認する場合の様子を図2に示す．左側のウインドウは検索元プログラム，右側のウインドウは検索先プログラムの逆アセンブリをIDA Proで開いた画面であり，検索先プログラムの下部にある Output Window 上に実行結果を表示する．表示された関数名をダブルクリックすると上部に該当する関数が表示されるため，解析者は速やかに関数の内容を確認できる．

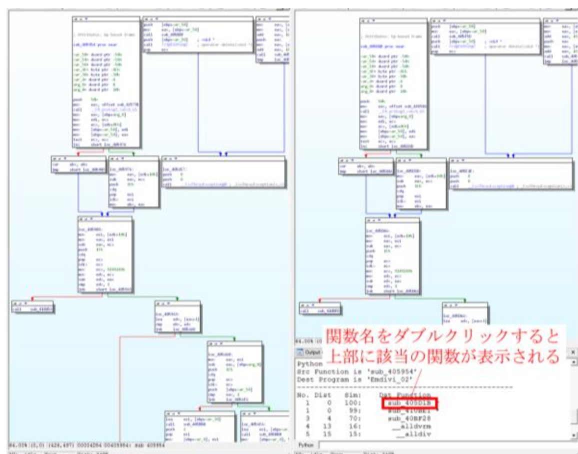

図2　提案方式の実行例（GUI）
Figure 2　Execution example of proposal method (GUI).

5. 評価

提案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．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，メモリ一容量8GB，SSD 256GB，Windows 8 64bitである．

5.1 GNU bash による評価

GNU bash の2つのバージョンのプログラム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．2つのプログラムに対して Linux の strip コマンドでシンボル情報を削除した上で，bash 4.0 でシンボルが削除された 381 個の関数に対して提案方式を用いて bash 4.3.30 の関数を検索し，同じシンボル情報を持っていた関数が検索結果の上位10位以内に表示されるか評価した．

期待する関数が検索結果に表示された順位の統計を図3に示す．314個の関数は1位として出力され，38個の関数が10位以下として出力された．

提案方式をBinDiffと比較した結果を表2に示す．BinDiffは345個の関数を正解し，提案方式は343個の関数を正解した．全体としてほぼ同精度の精度を達成した．要件として，BinDiffが不正解であった関数に対する提案方式の適用を想定しており，BinDiffが不正解となった36個の関数のうち29個（80.6%）に対して正解したため，BinDiffと組み合わせることで全体の正解範囲が広がり，提案方式の有用性が示せた．

表1　比較に使用した bash のバージョン
Table 1　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

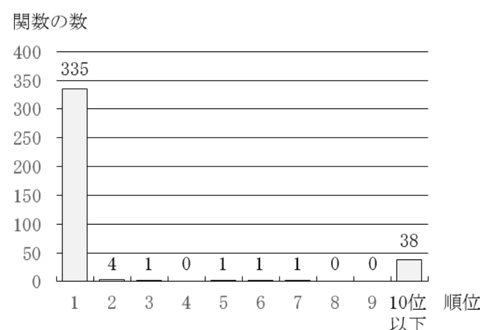

図3　bash における提案方式の検索順位
Figure 3　Rank result of bash.

表2　bash における提案方式と BinDiff の比較評価
Table 2　Experiment result of bash comparison.

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

提案方式とBinDiffの実行時間を評価した．BinDiffは全ての関数に対して3.7秒で処理を完了した．提案方式は1つの関数に対して平均0.9秒，最大19秒，全ての関数の検索に対して339.1秒という結果となった．BinDiffは関数の呼び出しを辿り連鎖的に対応付けるため全体として高速に動作した．提案方式は検索する関数に対して全ての関数との編集距離を計算するため，関数1つあたりの計算時間は高速となったが，全体としてはBinDiffより時間を要した．

5.2 GNU binutils による評価

GNU binutils の2つのバージョンのプログラム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．提案方式をBinDiffと比較した結果は表4の通り bash と同様の結果となった．

表3　比較に使用した binutils のバージョン
Table 3　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

表4　binutils における提案方式と BinDiff の比較評価
Table 4　Experiment result of binutils comparison.

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

提案方式と BinDiff の実行時間を評価した．BinDiff は全ての関数に対して 4.5 秒で処理を完了した．提案方式は 1 つの関数に対して平均 0.9 秒，最大 24.8 秒，全ての関数の検索に対して 1 プログラムあたり平均 640.1 秒であり, bash と同様の傾向を示した．

5.3 Emdivi による評価

マルウェア検体を用いて評価を行った．表 5 は一連の APT 攻撃で用いられた異なるバージョンの Emdivi という RAT である．最も古い検体 1 を解析したことがあるマルウェアとし，検体 2〜6 を解析するマルウェアと想定した．

検体 1 において検索元関数として使用した関数を表 6 に示す．この表ではこの関数の特徴を示すため，基本ブロック（BB）の数，命令数，主な処理内容を記載している．C&C 通信はファイアウォールのアクセス制御を回避するため一般的に HTTP が使用される．今回の評価に使用した関数は HTTP のリクエスト文を構築する処理と暗号化を行う処理である．インシデント対応では影響範囲を調査するため，プロキシサーバーに記録されたログを分析して他の感染端末や通信内容を特定する．これらの関数を静的解析することはログを調査する上で重要な意味を持つ．

表 5　評価に使用した Emdivi 検体
Table 5　Emdivi samples for evaluation.

No.	検体名	評価の役割	関数の数
検体 1	Emdivi t17.08.21	既知マルウェア	1018
検体 2	Emdivi t17.08.26	解析対象	1009
検体 3	Emdivi t17.08.30	解析対象	957
検体 4	Emdivi t17.08.30	解析対象	949
検体 5	Emdivi t17.08.31	解析対象	1145
検体 6	Emdivi t17.08.34	解析対象	1100

表 6　評価に使用した検体 1 の関数
Table 6　Function name of sample 1 for evaluation.

関数	BB 数	命令数	関数内で行う主な処理
sub_40801C	119	1026	POST リクエスト文の構築
sub_40CA5E	28	275	GET リクエスト文の構築
sub_40204D	13	157	XXTEA による暗号化

検体 1 の 3 つの関数に対して BinDiff と提案方式で比較を行った結果を表 7 に示す．このセルは，左側が BinDiff の成否，右側が提案方式の成否と検索順位を示す．バージョンが近い検体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．その他では，BinDiff が不正解を出力したものや，BinDiff と提案方式ともに不正解となったものがあった. BinDiff が不正解を出力した場合，これまでは情報が全くなく最初からコードを読み解く必要があったが, 提案方式では失敗しても 10 位以下の関数を順番に調べていくことができる．

実行時間については，BinDiff は全体で 10.2 秒，提案方式は関数あたり最大で 1.7 秒であり，解析者が手動で検索することを想定すると十分な速度であった．

表 7　BinDiff と提案方式による評価結果
Table 7　BinDiff and proposing method evaluation result.

検体 1 の関数	検体 2	検体 3	検体 4	検体 5	検体 6
sub_40801C	○,○1st	○,○1st	○,○3rd	×,○2nd	×,×15th
sub_40CA5E	○,○1st	○,○1st	×,○7th	×,×30th	×,×84th
sub_40204D	○,○1st	○,○1st	○,○1st	○,○1st	○,○1st

提案方式が正解を上位に出力できなかった関数は，関数内の制御フローグラフが大きく変化していたため正しく判定できなかった．逆に，提案方式が正解を出力していた関数は関数が大きく変化していなかった．提案方式が正解した中に BinDiff が誤答を出力したものがあったが，呼出元となる関数における制御フローグラフが大きく変化して正しく判定できず，そのため連鎖的に判断を誤っていた．

5.4 Emdivi と PlugX における全関数の評価

5.3 節における Emdivi 検体 1，2 と実際の APT で使用された PlugX と呼ばれる表 8 の RAT に対して，BinDiff と提案方式を用いて全ての関数の比較を行った結果を表 9，表 10 に示す．また，非常に大きい画像であるため一部となるが，可視化したグラフを図 4，図 5 に示す．この図は検体 1 のコールグラフであり，頂点は関数を意味する．BinDiff と提案方式を検体 1 の全ての関数に適用して目視で正解かどうかを判定した．正解かどうか判断できないものはともに不正解とした．たとえ正解であっても実際に解析する際に正解が判断できないと意味をなさないためである．評価結果は頂点の色で表現している．青色は BinDiff のみが正解したもの，赤色は提案方式のみが正解したもの，紫色は両方とも正解したもの，灰色は両方とも失敗したものである．この検体における関数呼び出しは非常に多く，全ての辺を記載すると現実的な図とならなかったため，各頂点に対して 1 辺だけをランダムに抜粋している．

BinDiff は関数の呼び出しを辿り連鎖的に対応付けを行うため，コールグラフにおいて失敗した頂点が固まる傾向があった．一方で提案方式は関数呼び出しの関係に依存しないため，失敗した頂点は散らばった．

関数呼び出しは call 命令によって行われるが，ジャンプ先のアドレスが動的に計算される場合がある．このような場合はコールグラフにおいて辺を設定することができず，独立した頂点となる．このような関数において BinDiff は呼び出し関係を利用することができないため失敗が目立っていた．

表 8　評価に使用した Plugx 検体
Table 8　PlugX samples for evaluation.

No.	検体名	評価の役割	関数の数
検体 7	PlugX	既知マルウェア	472
検体 8	PlugX	解析対象	517

表 9　Emdivi における提案方式と BinDiff の比較評価
Table 9　Experiment result of Emdivi comparison.

		BinDiff		合計
		正解	不正解	
提案方式	正解	725	40	765
	不正解	3	250	253
合計		728	290	1018

表 10　PlugX における提案方式と BinDiff の比較評価
Table 10　Experiment result of PlugX comparison.

		BinDiff		合計
		正解	不正解	
提案方式	正解	382	18	400
	不正解	43	29	72
合計		425	47	472

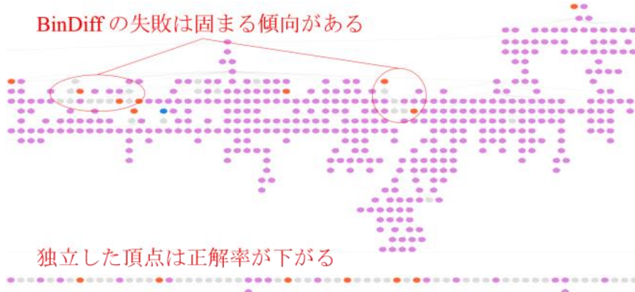

図 4　Emdivi の全関数に対する評価結果（一部）
Image 4　BinDiff and proposing method result for Emdivi

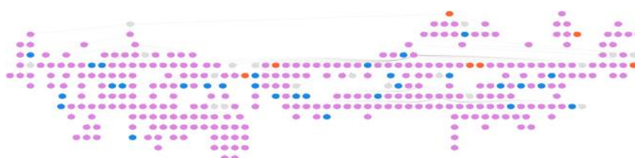

図 5　PlugX の全関数に対する評価結果（一部）
Image 5　BinDiff and proposal method result for PlugX

6. まとめと今後の課題

本研究では，過去に解析したマルウェアの関数の制御フローグラフと命令列を用いて相当する関数を検索するアルゴリズムを提案した．GNU bash と binutils の 11049 個の関数の 90.3% で正解を出力し，Emdivi の 6 個の検体を使用して，インシデント対応で解析が必要となる関数に対して有効性を示した．さらに Emdivi と PlugX の検体における全関数を比較して，BinDiff と提案方式の得意とするケースを評価し，併用することで正解率を向上できることを示した．

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).

[38] Cython: C-Extention for Python (online), available from〈http://cython.org/〉(accessed 2016-08-11).