

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,a)} 後藤 厚宏¹

受付日 2016年8月24日, 採録日 2017年2月9日

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。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た場合、過去に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。このコード特定のために、BinDiff に代表されるパッチ解析用のコード「比較」ツールを利用できるが、貪欲アルゴリズムにより対応付けを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。マルウェア解析に適したコード比較アルゴリズムとして、関数における制御フローグラフの編集距離と命令列の最長共通部分列を用いて関数を「検索」する BinGrep を提案する。BinGrep は、GNU bash と binutils では 11,049 個の関数のうち 90.3% について正解を出力できた。実際に APT 攻撃で使用されたマルウェアで評価したところ、Emdivi の 11 検体の評価では、インシデント対応において重要であった 27 個の関数の 85% について正解を出力できた。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,a)} ATSUHIRO GOTO¹

Received: August 24, 2016, Accepted: February 9, 2017

Abstract: In recent years, many Japanese organizations suffered a large-scale damage caused by APT activities. Prompt and appropriate incident responses are indispensable. When resemble malware is used in some attacks commonly, the analyst can proceed to static analysis immediately specifying the position of function previously analyzed. We can use code “comparison” tools such as BinDiff to specify this program code, but there are some problems that the method mistakes the matching continuously because of the greedy algorithm, and there are no information against such functions. As the algorithm which is suitable for malware analysis, we propose BinGrep method that “searches” for function using both the edit distance of control flow graph of functions and longest common substrings of instructions. We evaluated that 90.3% of the 11,049 functions of the GNU bash and binutils as normal program and 85% of the 27 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

¹ 情報セキュリティ大学院大学
Institute of Information Security, Yokohama, Kanagawa
221-0835, Japan

² NTT セキュリティ・ジャパン株式会社
NTT Security (Japan) KK, Chiyoda, Tokyo 101-0021, Japan

a) dgs158102@iisec.ac.jp

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。

APT 攻撃のように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であり、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いる [4]. インシデント対応におけるフォレンジックでは、この「検査」工程においてマルウェアの通信先や通信方式などの挙動を解析し、「分析」工程においてプロキシサーバなどのネットワークログと照合することで影響範囲を調査することがある。

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を受けて初めて動作するマルウェアを完全に解明することは難しい。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。たとえば、JPCERT/CC が APT 攻撃のログ分析手法の事例を紹介しているように、プロキシサーバのログから C&C 通信を調査することがあるため [6], 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。

JPCERT/CC によると、すべての手口が標的組織ごとに開発されるのではなく、類似の手口や攻撃インフラ (C&C サーバやマルウェア設置サイト、マルウェアなど) が利用されることも少なくないため、これらの情報が共有されることが望ましいとしている [7]. 特に、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。

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る研究が行われている [13], [14] (2.2 節参照)。これを実現する代表的な実装として、BinDiff というソフトウェアがある [15]. BinDiff は逆アセンブリ中に含まれるすべ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

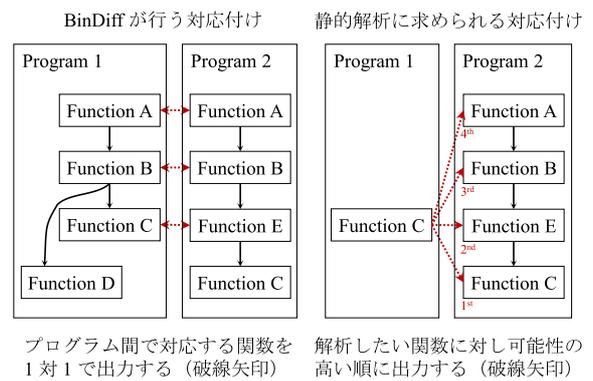


図 1 BinDiff の対応付けと静的解析で求められる要件
Fig. 1 BinDiff matching method and its requirement in static analysis.

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目的のコードを特定するうえで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解析者が利用できる情報が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はコストが高いため、緊急を要するインシデント対応においては、通信や暗号に関する処理など特定の挙動に着目して解析を行う。2 つのプログラム全体の対応や変更点を抽出する必要はなく、目的とする関数に対応付けられれば十分である。図 1 のプログラム 1, 2 における関数間の呼び出し関係を有向グラフで表現したコールグラフにおいて、BinDiff はプログラムを比較して対応する関数を出力するが、プログラム 1 の関数 C のように誤って対応付けする場合がある。関数 C を解析したい場合、関数 C に相当する可能性の高い関数から順に出力する方法が求められる。

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と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、11,049 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 1,069 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 11 個の検体を用いて、プロキシサーバのログ調査に必要となる HTTP リクエストを構成する処理と暗号処理を行う 27 個の関数を評価した結果、85% 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 1,018 個の関数のうち 75%、PlugX の 472 個の関数のうち 90% 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を効率化するため、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、暗号鍵の組合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 [8]。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の特定手法を提案している [9]。

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 [10], [11], [12]。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。これらをすべ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、意味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 [13]。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こと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 [14]。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 [15]。Bourquin らは、BinDiff が対応付けできなかった残りの関数の集合に対して、さらに拡張割当て問題のアルゴリズムを利用して対応付けを行う BinSlayer という手法を提案している [16]。Gao らは、最大共通部分グラフ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 [17]。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、tthttpd と gzip におけるパッチの修正箇所を特定している [18]。そのほかにも、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 [19], [20], [21], [22]。

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うえ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 [23]。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 [24]。

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から式木で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 [25]。中島らは、機械語命令列の正規化を行ったうえで、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 [26]。

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 [27]。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 [28]。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 [29]。

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ことに着目し、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 [30]。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 [31]。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 [32]。

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、解析すべき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を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解析者にとって無意味な出力である。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など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実行コードが変化しやすい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と課題

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、速やかに作業にと

りかかることができる。2.2 節で取り上げた BinDiff は、2つのプログラムの「比較」の機能を有するが、マルウェア解析において目的コードを特定する方式としては課題がある。本章では、BinDiff のアルゴリズム自体にその課題が内在することを示す。

BinDiff は逆アセンブラの IDA Pro [38] 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、この単位の逆アセンブリ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 $(\alpha, \beta, \gamma) = (\text{基本ブロックの数, 基本ブロック間の辺の数, 関数呼び出しの数})$ という 3次元の特徴量を与える。

比較される 2つのプログラム A, B における関数の集合、すなわちコールグラフの頂点の集合を $G_A = \{a_1, \dots, a_n\}$, $G_B = \{b_1, \dots, b_m\}$ とし、その部分集合を $S_A \subseteq G_A$, $S_B \subseteq G_B$ とする。BinDiff は Selector と Property という関数を用いる。Selector ε は $a_i \in S_A$ と S_B を入力として、 a_i に一致する S_B の単独の要素を返す関数である。ただし、候補が複数存在する場合は何も返さない。「一致する」とは様々なアイデアが考えられるが、たとえば (α, β, γ) が一致する、命令列から生成した CRC32 が一致する、関数名が一致する、など複数の Selector を定義して利用する。Property π は $a_i \in S_A$ と S_A もしくは $b_j \in S_B$ と S_B を入力として、 a_i や b_j と関係の深い頂点の集合 S'_A や S'_B を出力する。たとえば、頂点に対する呼び出し元となる頂点の集合を返す関数、頂点から呼び出す先の頂点の集合を返す関数がある。

アルゴリズム 1 の initialMatches で最初の対応付けを行

アルゴリズム 1 initialMatches

Algorithm 1 initialMatches.

In	S_A : プログラム A に含まれる関数の集合 S_B : プログラム B に含まれる関数の集合
Out	M: プログラム A と B の間に対応する関数のリスト S_A : 入力 S_A に対し対応付けが残った関数の集合 S_B : 入力 S_B に対し対応付けが残った関数の集合
1	Function initialMatches (S_A, S_B)
2	$M \leftarrow \emptyset$
3	foreach $a_i \in S_A$ do
4	foreach Selector ε do
5	if $(a_i, b_j) \leftarrow \varepsilon(a_i, S_B)$ then
6	$M \leftarrow M \cup \{a_i \mapsto b_j\}$
7	$S_A \leftarrow S_A \setminus \{a_i\}$
8	$S_B \leftarrow S_B \setminus \{b_j\}$
9	break
10	return (M, S_A, S_B)

アルゴリズム 2 propagateMatches
Algorithm 2 propagateMatches.

In	M: プログラム A と B の間に対応する関数のリスト S _A : プログラム A に含まれる関数の集合 S _B : プログラム B に含まれる関数の集合
Out	M: 入力 M に対し対応付けを拡大した関数のリスト S _A : 入力 S _A に対し対応付けが残った関数の集合 S _B : 入力 S _B に対し対応付けが残った関数の集合
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 a' _i ∈ S' _A do
8	foreach Selector ε do
9	if (a' _i , b' _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.

In	G _A : プログラム A における関数の集合 G _B : プログラム B における関数の集合
Out	M: 対応付けされた関数のリスト S _A : プログラム A にのみ含まれる関数の集合 S _B : プログラム B にのみ含まれる関数の集合
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)

う。プログラム A と B における関数の集合 S_A, S_B を入力として、すべての頂点 a_i ∈ S_A に対しすべての Selector を適用し、その出力 b_j に対して対応 {a_i ↦ b_j} をリスト M の中に記録する。対応付けられた頂点は順次 S_A, S_B の中から除外する。

アルゴリズム 2 の propagateMatches はすでに対応付けされた頂点のリスト M をもとに、対応付けの範囲を拡大する。すべての対応付け {a_i ↦ b_j} ∈ M に対して a_i, b_j に Property を適用して関連する頂点の集合を取得し、再びこの範囲で initialMatches と同様にすべての Selector を適用して対応をリスト M に記録する。

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 次元の特徴量を取得し、アルゴリズム 3 の binDiff を実行する。この中で、initialMatches を実行して初期の対応付けを求め、停止するまで繰り返し propagateMatches を実行する。

BinDiff のアルゴリズムについて、図 2 のコールグラフ

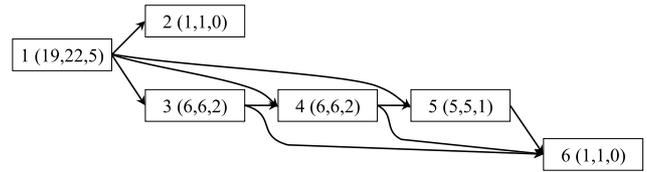


図 2 BinDiff アルゴリズムを実行するコールグラフ
Fig. 2 Callgraph example for BinDiff algorithm.

を持つプログラムを例に、自身のプログラムと比較することで実際の動作を示す。頂点のラベルはノードの番号と特徴量 (α, β, γ) である。この例で使用される Selector は特徴量が一致するものを選ぶ関数とし、propagateMatches は呼び出し元、呼び出し先の頂点の集合を返す 2 つの関数とする。最初に InitialMatches を実行すると、Selector によって頂点 1, 5 がユニークに対応付けられる。次に 1 回目の propagateMatches において、Property を頂点 1 に適用して呼び出し先の頂点 {2, 3, 4} を取得してこの中で再度 Selector を実行すると、特徴量がユニークである頂点 2 を対応付けられる。また、頂点 5 の呼び出し元、呼び出し先の頂点 {4, 6} の中でそれぞれ Selector を実行すると、頂点 4 と頂点 6 の両方が対応付けられる。propagateMatches の 2 回目の実行では、同様に頂点 3 を対応付ける。

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すべ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たど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うえで十分な情報が得られない場合がある。

4. 提案方式 BinGrep

4.1 要件

本節では、インシデント対応におけるマルウェア解析に適したコード比較方式 (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 に適用してすべ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。解析者が誤った出力であると判断した場合に提案方式を適用する。解析者は、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して、マルウェア亜種における関数を「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と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に、逆アセンブリを並べて比較できるインタ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。

提案方式ではBinDiffを代表とする従来研究と同様に、逆アセンブリ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、外部関数名などの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

提案方式では、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したアルゴリズムを提案する。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 [33]。

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 [34]。

ラベル付き木に対する編集距離としては、Zhang らの計算量 n^4 のアルゴリズム [35] と Klein の計算量 $n^3 \log n$ の

アルゴリズム [36] 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 [37]。

4.4 提案アルゴリズム

本節では提案方式における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。提案アルゴリズムはアルゴリズム 4 の constructCFT とアルゴリズム 5 の binGrep で構成される。

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 $V = \{V_1, V_2, \dots, V_n\}$ 、辺を $E = \{(V_i, V_j), \dots\}$ (i, j は頂点 V のインデックス) と定義する。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。JumpTo(G, i) は V_i に対する関数呼び出し先の頂点の集合を返す関数、DelJump(G, i, j) は G の辺から (V_i, V_j) を削除する関数とする。setLabel(G, i) は G の頂点 V_i にラベルを与える関数とする。基本ブロック V_i において call 命令で外部関数を呼び出している場合、この外部関数名を昇順に連結した文字列をラベルとする。たとえば、「call memset」と「call printf」を実行する基本ブロックにおけるラベルは“memset printf”となる。外部関数呼び出しが存在しない場合は共通のラベル“NULL”を与え

アルゴリズム 4 constructCFT

Algorithm 4 constructCFT.

In	G : 制御フローグラフ i : G の頂点 (基本ブロック) 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 $j \in \text{JumpTo}(G, i)$ do
3	if $j \in m$ do
4	$G \leftarrow \text{DelJump}(G, i, j)$
5	else
6	$m \leftarrow m \cup \{j\}$
7	$G \leftarrow \text{setLabel}(G, j)$
8	$G \leftarrow \text{constructCFT}(G, j, m)$
9	return G

アルゴリズム 5 binGrep

Algorithm 5 binGrep.

In	G_A : プログラム A における検索元となる関数 G_B^* : プログラム B の関数 G_B の集合
Out	r' : 検索結果のリスト
1	function binGrep(G_A, G_B^*)
2	$r \leftarrow \emptyset$
3	setLabel(G_A, \emptyset)
4	constructCFT($G_A, \emptyset, \emptyset$)
5	foreach $G_B \in G_B^*$ do
6	setLabel(G_B, \emptyset)
7	constructCFT($G_B, \emptyset, \emptyset$)
8	$r \leftarrow r \cup (\text{EditDistance}(G_A, G_B), \text{LCS}(G_A, G_B))$
9	$r' \leftarrow \text{Sort}(r)$
10	return r'

```

$ 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 3 提案アルゴリズムの実行例 (CUI)

Fig. 3 Execution example of proposing method (CUI).

る. constructCFT は再帰関数として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^* のすべての要素と比較し, 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うえで $EditDistance(G_A, G_B)$ を用いて 2 つの制御フローグラフ間における編集距離を計算し, さらに $LCS(G_A, G_B)$ を用いて関数 G_A, G_B に含まれる命令列のオペコードに対して最長共通部分列を計算し, これらの結果を 2 次元ベクトルとして r に格納する. 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 らのアルゴリズムを実装した Python ライブラリ [39] をベースに, Python を C 言語拡張する Cython [40] に移植して高速化したものを使用した.

提案アルゴリズムの実行例を示す. 図 3 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) が大きい順に並び, 最小の編集距離を持つ関数名の右には「*」印が付与される.

マルウェア解析者が正解を確認する場合の様子を図 4 に

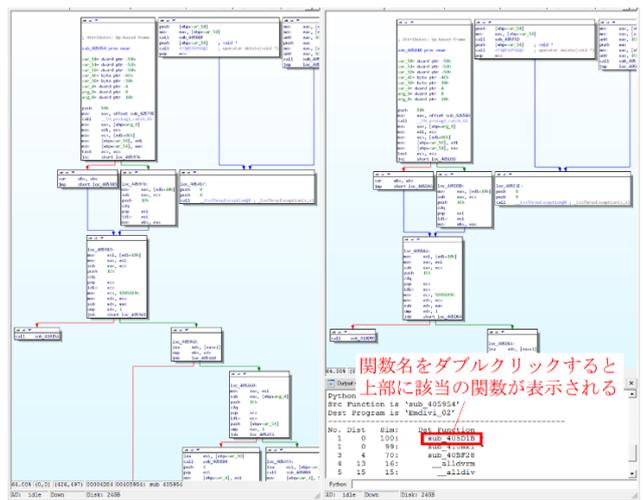


図 4 提案アルゴリズムの実行例 (GUI)

Fig. 4 Execution example of proposing method (GUI).

示す. 左側のウィンドウは検索元プログラム, 右側のウィンドウは検索先プログラムを 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の逆アセンブリが表示されるため, 解析者は速やかに関数の内容を確認できる.

5. 評価

提案方式 (BinGrep) 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. 正規のプログラムでは 2 つのバージョンの GNU bash と binutils においてすべての関数を検索し BinDiff の結果と比較した. マルウェア検体においては, 6 つの Emdivi を使用し, インシデント対応で重要となった 3 つの関数について検索を行い, BinDiff の結果と比較した. さらに, Emdivi と PlugX のそれぞれ 2 つの検体を使用し, すべての関数を検索して BinDiff の結果と比較した.

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 1.20 GHz, メモリ容量 8 GB, SSD 256 GB, Windows 8 64 bit 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 1 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うえで, 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 の関数を検索した. 同じシンボル情報を持つ関数を正解と定義し, BinDiff が正解を出力できるかどうか, 提案方式における検索結果の上位 10 位以内に正解が出力されるか

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

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		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

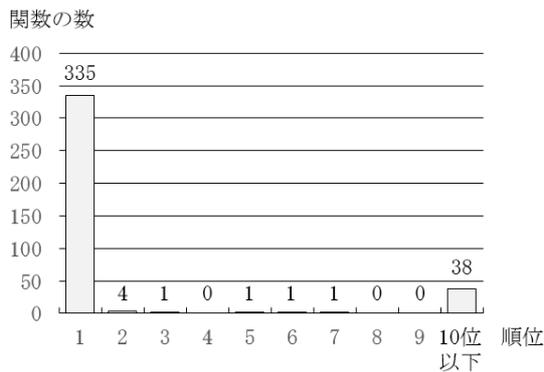


図 5 bash における提案方式の検索順位

Fig. 5 Rank result of bash.

どうか、それぞれ評価を行った。

提案方式と BinDiff の比較結果を表 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。全体としては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された。

また、提案方式において正解が検索結果に表示された順位の統計を図 5 に示す。335 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。

提案方式と BinDiff の実行時間を評価した。BinDiff はすべての関数に対して 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、すべ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たどり連鎖的に対応付けを行う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すべ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。

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		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

表 5 評価に使用した Emdivi 検体

Table 5 Emdivi samples for evaluation.

検体 No.	検体名	関数の数
検体 1	Emdivi t17.08.21	1018
検体 2	Emdivi t17.08.26	1009
検体 3	Emdivi t17.08.30	957
検体 4	Emdivi t17.08.30	949
検体 5	Emdivi t17.08.31	1145
検体 6	Emdivi t17.08.34	1100
検体 7	Emdivi t20.02	1079
検体 8	Emdivi t20.03	1543
検体 9	Emdivi t20.03	1035
検体 10	Emdivi t20.06	1204
検体 11	Emdivi t20.08	2308

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 2 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。

bash と同様に、binutils 2.22 でシンボルが削除された 10,668 個の関数に対して提案方式を用いて binutils 2.25.1 の関数を検索した。提案方式を BinDiff と比較した結果を表 4 に示す。BinDiff は 9,635 個の関数を正解し、提案方式は 9,651 個の関数を正解した。特に、BinDiff が不正解となった 1,033 個の関数のうち 674 個 (65.2%) に対して正解し、bash と同様の評価結果となった。

提案方式と BinDiff の実行時間を評価した。BinDiff はすべての関数に対して 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、すべ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

表 6 評価における Emdivi 検体の役割

Table 6 The role of Emdivi samples under evaluation.

評価 No.	既知マルウェア	解析対象
評価 3	検体 1 (t17 系)	検体 2~6 (t17 系)
評価 4	検体 7 (t20 系)	検体 8~11 (t20 系)
評価 5	検体 1 (t17 系)	検体 7~11 (t20 系)

表 7 評価に使用した検体 1 の関数

Table 7 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

表 8 評価に使用した検体 7 の関数

Table 8 Function name of sample 7 for evaluation.

関数	BB 数	命令数	関数内で行う主な処理
sub_415FB1	185	1254	POST リクエスト文の構築
sub_414808	72	698	GET リクエスト文の構築
sub_40113A	45	263	AES による暗号化

APT 攻撃で用いられた Emdivi という RAT である。このマルウェアにはバージョンがハードコードされており、検体 1~6 は t17、検体 7~11 は t20 というバージョンを持つ。これらのマルウェアに対して、表 6 に示す 3 つの状況を想定して評価 3~5 を行った。

検体 1、検体 7 において検索元関数として使用した関数を表 7、表 8 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うえで重要な意味を持つ。

検体 1 と検体 7 における各 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 9、表 10 に示す。セルの左側は、BinDiff が出力した解答に対して正解か不正解かを「○」「×」で示したものである。右側は、提案方式において正解となるべき関数が 10 位以内に表示されたかどうかを「○」「×」で示したものであり、さらに正解が出力された順位を記載している。正解となるべき関数がプログラムに存在しない場合は「×無」と記載している。

評価 3 の結果を表 9 に示す。バージョンが近いほど BinDiff も提案方式も正解率が高くなる傾向がみられたが、検体 1 と検体 4、検体 1 と検体 5 では提案方式のみが上位に正解を出力できた関数が存在した。BinDiff が不正解を出力した場合、これまでは情報がまったくなく最初からコー

表 9 検体 1 における BinDiff と提案方式による評価結果

Table 9 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	○,○1 st	○,○1 st	○,○3 rd	×,○2 nd	×,×15 th
sub_40CA5E	○,○1 st	○,○1 st	×,○7 th	×,×30 th	×,×84 th
sub_40204D	○,○1 st	○,○1 st	○,○1 st	○,○1 st	○,○1 st

表 10 検体 7 における BinDiff と提案方式による評価結果

Table 10 BinDiff and proposing method evaluation result.

検体 7 の関数	検体 8	検体 9	検体 10	検体 11
sub_415FB1	○,○1 st	○,○1 st	○,○1 st	×,○1 st
sub_414808	×,○10 th	○,○1 st	○,○1 st	×,○2 nd
sub_40113A	○,○1 st	○,○1 st	○,○1 st	×,×無

表 11 検体 1 における BinDiff と提案方式による評価結果

Table 11 BinDiff and proposing method evaluation result.

検体 1 の関数	検体 7	検体 8	検体 9	検体 10	検体 11
sub_40801C	×,×無	×,×無	×,×無	×,×無	×,×無
sub_40CA5E	×,×無	×,×無	×,×無	×,×無	×,×無
sub_40204D	×,×無	×,×無	×,×無	×,×無	×,○1 st

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。

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、提案方式が正解を上位に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つたが、呼び出し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。

評価 4 の結果を表 10 に示す。検体 7 と検体 8、検体 7 と検体 11 において提案方式の優位性が認められた。検体 11 においては、検体 7 の sub_40113A に相当する AES 暗号化関数は存在しなかった。JPCERT/CC によると、t17 で使用された暗号化関数は XXTEA であり、t20 においては t20.6 以下は AES、t20.7 から t20.25 までは XXTEA が使用されていることが報告されている [41]。本評価ではこの報告と一致することが確認された。

評価 5 の結果を表 11 に示す。HTTP リクエスト文を構築する処理は実装が大きく変化しており、新たに開発されたコードであると推測された。このため正解は存在しなかった。ただし検体 11 では t17 と同じ XXTEA が使用されたため、正解を出力することができた。

実行時間については、BinDiff は全体で 11.0 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。

5.4 Emdivi と PlugX における全関数の評価

BinDiff と提案方式を用いてマルウェア検体におけるす

表 12 評価に使用した Plugx 検体
Table 12 PlugX samples for evaluation.

検体 No.	検体名	評価の役割	関数の数
検体 12	PlugX	既知マルウェア	472
検体 13	PlugX	解析対象	517
検体 14	PlugX	解析対象	517
検体 15	PlugX	解析対象	495
検体 16	PlugX	解析対象	495
検体 17	PlugX	解析対象	495
検体 18	PlugX	解析対象	495
検体 19	PlugX	解析対象	495
検体 20	PlugX	解析対象	472
検体 21	PlugX	解析対象	472

表 13 Emdivi における提案方式と BinDiff の比較評価
Table 13 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 14 PlugX における提案方式と BinDiff の比較評価 (平均)
Table 14 Experiment result of PlugX comparison (average).

		BinDiff		合計
		正解	不正解	
提案方式	正解	413.9	13.9	427.8
	不正解	25.8	18.4	44.2
合計		439.7	32.3	472.0

すべての関数の比較を行った。評価 6 では 5.3 節における Emdivi 検体 1, 2 を使用し、評価 7 では実際の APT で使用された PlugX と呼ばれる表 12 の RAT を使用した。その結果を表 13, 表 14 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 6, 図 7 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 のすべ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、すべ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たど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。

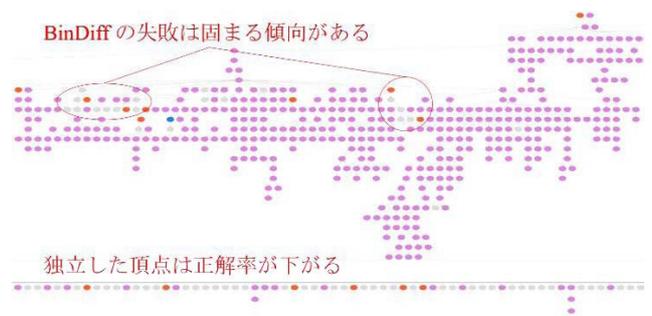


図 6 Emdivi の全関数に対する評価結果 (一部)
Fig. 6 BinDiff and proposing method result for Emdivi.

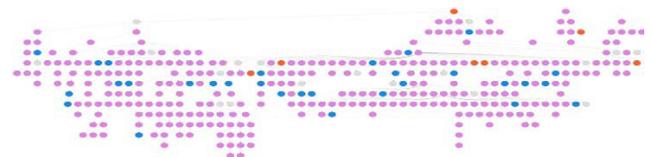


図 7 PlugX の全関数に対する評価結果 (一部)
Fig. 7 BinDiff and proposal method result for PlugX.

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、独立した関数となる。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。

6. 議論

解析者が手動で関数の検索を行う場合は逆アセンブリを先頭から読む必要があるため、相当の作業量がかかることが想定されるが、提案方式を使用すると解析者にとって負担にならない時間で目的の関数を特定することができる。インシデント対応におけるマルウェア解析に活用した場合、マルウェアの通信や暗号処理に関わる関数を特定して、その中の処理を過去のコードと比較して差分を解析するが、たとえば今回の評価に使用した Emdivi では解析者が一見するだけで過去のマルウェアと同じ暗号方式が使用されていることが特定できている。さらに、通信先や暗号方式が厳密に特定できるかどうかでインシデント対応の方針が大きく変わることも考えられる。

一方で、解析対象マルウェアが過去に解析したどのマルウェアに類似するかどうかは解析者が特定する必要がある。新たに開発されたマルウェアや機能については適用できない。また、関数構造を持たないマルウェアや極端に難読化されたマルウェアに対しても提案方式を適用できない。パッキングされたマルウェアや他のプログラムにインジェクトされた実行コードは、メモリから実行コードを取り出してから逆アセンブルしたうえで提案方式を適用する必要がある。

7. まとめと今後の課題

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、速やかに解析にとりかかることができる。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新しい方式 (BinGrep) を提案した。

正常プログラムによる評価では GNU bash と binutils の 11,049 個の関数の 90.3% について正解を出力した。Emdivi による評価では 11 個の検体を使用して、インシデント対応に必要な HTTP リクエストの構築と暗号化の処理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させることができることを示した。

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。また、本稿では評価プログラムとマルウェアにおいて関数をどれだけ正しく検索できるかを評価したが、提案方式を活用することで実際のインシデント対応における作業量や精度の改善にどれだけ貢献できるか評価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 2016-08-20).
- [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 2016-08-20).
- [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] JPCERT/CC：ログを活用した高度サイバー攻撃の早期発見と分析 (オンライン), 入手先 <https://www.jpCERT.or.jp/magazine/acreport-emdivi.html> (参照 2016-12-25).
- [7] JPCERT/CC：高度サイバー攻撃への対処におけるログの活用と分析方法 (オンライン), 入手先

- <https://www.jpCERT.or.jp/magazine/acreport-emdivi.html> (参照 2016-12-25).
- [8] 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).
 - [9] 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).
 - [10] Percival, C.: Naive differences of executable code, Binary diff/patch utility (online), available from <http://www.daemonology.net/papers/bsdifff.pdf> (accessed 2016-08-20).
 - [11] 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).
 - [12] Heirbaut, J.: Diff utility for binary files, JojoDiff (online), available from <http://jojodiff.sourceforge.net> (accessed 2016-08-20).
 - [13] Flake, H.: Structural Comparison of Executable Objects, *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2004)*, pp.161–173, IEEE Computer Society (2004).
 - [14] Dullien, T. and Rolles, R.: Graph-based comparison of Executable Objects, *Proc. SSTIC 2005* (2005).
 - [15] Zynamics: Zynamics BinDiff (online), available from <http://www.zynamics.com/bindiff.html> (accessed 2016-08-20).
 - [16] 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).
 - [17] 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).
 - [18] 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).
 - [19] Oh, J.: Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries, *Proc. Black Hat USA 2009* (2009).
 - [20] Tenable Network Security: PachDiff2 High Performance Patch Analysis (online), available from <https://www.tenable.com/blog/patchdiff2-high-performance-patch-analysis> (accessed 2016-08-20).
 - [21] 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 2016-08-20).
 - [22] Zimmer, D.: IDACmpare, VeriSign iDefense Labs (online), available from <http://sandsprite.com/iDef/IDACmpare/> (accessed 2016-08-20).
 - [23] 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).

- [24] 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).
- [25] 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).
- [26] 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).
- [27] 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).
- [28] Karim, M.E., Walenstein, A., Lakhotia, A. and Parida, L.: Malware phylogeny generation using permutations of code, *Journal of Computer Virology*, Vol.1, No.1-2, pp.13–23, Springer-Verlag (2005).
- [29] Gheorghescu, M.: An automated virus classification system, *Virus Bulletin Conference 2005* (2005).
- [30] 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).
- [31] 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).
- [32] Kinable, J. and Kostakis, O.: Malware Classification based on Call Graph Clustering, *Journal in Computer Virology*, Vol.7, No.4, pp.233–245, Springer-Verlag (2011).
- [33] 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).
- [34] Gao, X., Xiao, B., Tao, D. and Li, X.: A survey of graph edit distance, *Pattern Analysis and Applications*, Vol.13, No.1, pp.113–129, Springer-Verlag (2010).
- [35] Zhang, K. and Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems, *SIAM Journal of Computing*, Vol.18, No.6, pp.1245–1262 (1989).
- [36] Klein, P.: Computing the edit-distance between unrooted ordered trees, *Proc. 6th Annual European Symposium on Algorithms*, pp.91–102, Springer-Verlag (1998).
- [37] 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).
- [38] Hex-Rays: IDA Pro (online), available from <http://www.hex-rays.com> (accessed 2016-03-06).
- [39] Henderson, T. and Johnson, S.: Zhang-Shasha: Tree edit distance in Python, available from <https://zhang-shasha.readthedocs.org/en/latest/> (accessed 2016-08-20).
- [40] Cython: C-Extension for Python (online), available from <http://cython.org/> (accessed 2016-08-20).
- [41] JPCERT/CC : Emdivi が持つ暗号化された文字列の復号 (オンライン), 入手先 <https://www.jpCERT.or.jp/magazine/acreport-emdivi.html> (参照 2016-12-25).



羽田 大樹 (正会員)

2006年東京工業大学大学院修士課程修了。同年NTTコミュニケーションズ株式会社入社。2016年NTTセキュリティ・ジャパン株式会社出向, セキュリティログ分析業務に従事。2013年情報セキュリティ大学院大学修士課程修了。2015年同大学院博士課程入学。



後藤 厚宏 (正会員)

1984年東京大学大学院博士課程修了(工学博士)。NTT研究所にてクラウドセキュリティ技術等の研究開発に従事。2011年7月より情報セキュリティ大学院大学教授。情報処理学会理事。本会フェロー。