グラフ埋め込みを用いたIoTマルウェアの機能推定手法の改善の検討

高田 智史[1,a)]　川添 玲雄[1]　何 天祥[1]　韓 燦洙[2]　田中 智[2]　竹内 純一[1]

概要：脆弱なIoT機器に感染するIoTマルウェアの数が増大している．これらの多くは一般に公開されているソースコードの一部を改変して機能の追加や変更，削除を行うことで作成された亜種と呼ばれるものである．先行研究で我々は，亜種が持つ機能を推定する目的で，検体が呼び出す関数の時系列情報を有向グラフ化し，シグネチャを用いて検体の機能を推定する手法を提案した．ここでシグネチャはマルウェアの典型的な各機能に対応する部分グラフである．しかし，この手法ではシグネチャと完全にマッチしないが，機能の観点では同等と見做せるものを検体から検出できない．本稿では，シグネチャ内に僅かな関数の追加が行われているだけで，マッチさせる必要性が高い検体を特定する手法を提案し，その存在を確認した．また，それら柔軟なシグネチャマッチが必要な検体の候補を列挙するグラフ埋め込み手法を提案する．さらに，シグネチャの完全マッチ手法とクラスタリング手法の結果を横断的に分析することで，クラスタ内の検体における各シグネチャのマッチ数に傾向があることを確認した．

キーワード：IoTマルウェア，マルウェア解析，シグネチャマッチング，グラフ埋め込み，クラスタリング

Study on Improvement of Function Estimation Method of IoT Malware Using Graph Embedding

Satoshi Takada[1,a)]　Reo Kawasoe[1]　Tianxiang He[1]　Chansu Han[2]　Akira Tanaka[2]
Jun'ichi Takeuchi[1]

Abstract: Most IoT Malware is variants generated by editing and reusing a part of the functions based on publicly available source codes. In our previous study, we proposed a method to estimate the functions of a specimen using a signature, which is a directed graph of time series information of function calls. However, this method cannot detect those that perfectly match the signature but can be regarded as equivalent in terms of function. In this paper, we confirm the existence of specimens that have only a few additional functions in signature and propose a graph embedding method that enumerates the candidates of specimens that need more flexible signature matching. In addition, by cross-cutting analysis of results of the perfect match of signature and clustering method, we confirmed that there is a trend in the number of matches for each signatures in the specimens in the cluster.

Keywords: IoT malware, malware analysis, signature matching, graph embedding, clustering

1. はじめに

近年のIoT機器の増加に伴い，脆弱な機器に感染するIoTマルウェアの数が増大している．IoTマルウェアには，それぞれに固有の攻撃手法を持つ複数のファミリーが存在している．その中でも，BashliteやMiraiといった代表的

[1] 九州大学
Kyushu University
[2] 国立研究開発法人情報通信研究機構
National Institute of Information and Communications Technology
[a)] takada@me.inf.kyushu-u.ac.jp

なマルウェアファミリーにおいては，それらのソースコードが一般に公開 [5], [6] されており，第三者がこのソースコードを使って，機能の追加や変更・削除を行うことで一部の機能を改変した新しいマルウェアが作成可能である．このように作成されたマルウェアは亜種と呼ばれている．

しかしながら，AVClass [9] などによるマルウェア科名判定では，Bashlite や Mirai といった代表的なマルウェア科名しか得ることができない．同じマルウェア科名を持つ検体の間にも機能の違いはあると考えられ，科名判定だけではどのような機能を有する亜種が流行しているのか知ることは難しい．そこで我々の先行研究で Kawasoe らは，マルウェア科名以上の詳細な分析を行うことを目的に，Function Call Sequence Graph (FCSG) を用いたシグネチャマッチングによる検体の機能推定手法を提案した [7]．FCSG は検体が機能を実現するために呼び出す関数の情報を有向グラフ化したものであり，手動で作成したマルウェアの典型的な機能に対応したシグネチャとなる FCSG(以下，シグネチャ-FCSG と呼ぶ) が検体の FCSG に部分グラフとして含まれるかグラフマッチングを実施している．

しかし，グラフマッチングではシグネチャの部分グラフの完全一致を判定しており，シグネチャ内に関数の追加等が一つでもあれば，機能の観点では大きな変化は無いと考えられるにも関わらず，そのシグネチャがマッチしなくなってしまう．

そこで，先行研究 [7] よりも柔軟なシグネチャマッチングを行うことを目的に，グラフ埋め込みを用いたマルウェアの機能推定手法の改善について検討する．グラフ埋め込みとは，グラフの構造的特徴を捉えた上でグラフを実ベクトルで表現すること手法のことである．FCSG にグラフ埋め込みを用いることで，検体の FCSG 内にシグネチャ-FCSG が含まれるか否かの 2 値判定ではなく，シグネチャ-FCSG との類似度を実数値で出力することでき，先行研究 [7] よりも柔軟にシグネチャマッチングを行うことが期待できる．

本稿では，実際に先行研究 [7] で作成したシグネチャにおいて，マッチさせることが出来なかった検体のうち，シグネチャ内に少数の関数の追加が行われいるだけの検体，すなわち，そのシグネチャに類似した機能を持ち，マッチさせる必要があると思われる検体の存在を確認した．そのうえで，グラフ埋め込みを用いた，より柔軟なシグネチャマッチング手法を検討する．さらに，He ら [2] の正規化圧縮距離と高速近似法を用いたマルウェアクラスタリング手法において，各クラスタ内に属する検体に先行研究 [7] でのシグネチャマッチングを実施することで，生成されたクラスタ内の検体における各シグネチャのマッチ数に傾向があることを確認した．

本稿における貢献は以下の通りである．

- 先行研究 [7] で作成したシグネチャのうち，あるシグネチャ内に少数の関数追加が行われ，機能面で大きな変化は無いが，マッチさせることができていない検体の存在の確認
- グラフ埋め込みを用いた柔軟なシグネチャマッチング手法の提案
- He ら [2] のマルウェアクラスタリング手法によって生成された各クラスタ内の検体に，先行研究 [7] でのシグネチャマッチングを適用し，生成されたクラスタ内の検体における各シグネチャのマッチ数に傾向があることを確認

本稿の構成は以下の通りである．第 2 節で事前知識について説明した後，第 3 節で先行研究について説明する．第 4 節では提案手法について説明する．第 5 節では行った実験等で得られた結果等について報告し，第 6 節でまとめと今後の課題を述べる．

2. 事前知識

この節では，本稿で必要な基礎知識について説明する．

2.1 命令セットアーキテクチャ (ISA)

IoT 機器に用いられている CPU には様々なものが存在し，それぞれに固有の実行可能な命令集合である命令セットアーキテクチャ (ISA) が定められている．IoT 機器を狙うマルウェアは，ARM や MIPS，x86 など様々な CPU で動作できるように，複数の ISA 向けにコンパイルされている．また，たとえ同一のソースコードから生成したバイナリファイルでも，CPU すなわち ISA が異なると，その ISA 上で動作するバイナリの構造は異なるため，単純なバイナリの比較は行うことが出来ない．

2.2 逆アセンブル

プログラムは C 言語などの高級言語で人間によってソースコードとして書かれた後，コンパイラによって CPU が理解し実行可能な機械語の列であるバイナリファイルに変換される．一般的にマルウェアを解析する場合，解析者は元のソースコードを入手できない場合が多い．逆アセンブルは，バイナリファイルを人間が理解可能なアセンブリ言語に変換する処理のことで，マルウェアを静的解析する場合によく用いられる．IDA Pro [4] は Hex Rays が開発した逆アセンブルツールの一つであり，入力したバイナリファイルを対応するアセンブリ言語に変換する．また，関数毎にアセンブリ言語の制御フローを生成することができる．

2.3 Graph2vec

今回，実験で使った Graph2vec [10] というグラフ埋め込み手法について紹介する．Graph2vec はノードラベル付きのグラフ集合 $G = \{G_1, G_2, \cdots, G_N\}$ に対し，教師なしでグラフ全体の特徴ベクトルを学習する手法である．

Graph2vec では，まず個々のグラフ G_i を根付き部分グ

ラフの集合として表現する．事前に根付き部分グラフの最大の深さ D を設定して，各ノードを根として，Weisfeiler-Lehman [12] 再ラベル操作の繰り返しにより、深さが 0 から D までの根付き部分グラフを作る．その後，部分グラフから構成されたグラフに skip-gram モデル [8] を適用する．グラフ集合 $G = \{G_1, G_2, \cdots, G_N\}$ とグラフ G_i から抽出された根付き部分グラフの集合 $\{sg_1, sg_2, \cdots, sg_{l_i}\}$ に対して，G_i と sg_j の δ 次元の特徴ベクトル $\overrightarrow{G_i} \in \mathbb{R}^\delta$ と $\overrightarrow{sg_j} \in \mathbb{R}^\delta$ を学習する．学習モデルは，sg_j が G_i から抽出された条件に基づき，式 1 で表される対数尤度関数の最大化を目指す．

$$\sum_{j=1}^{l_i} \log \Pr(sg_j|G_i) \tag{1}$$

ここで条件付き確率 $\Pr(sg_j|G_i)$ は次のように定義される．

$$\Pr(sg_j|G_i) = \frac{\exp(\overrightarrow{G_i} \cdot \overrightarrow{sg_j})}{\sum_{sg \in \mathrm{Voc}} \exp(\overrightarrow{G_i} \cdot \overrightarrow{sg})} \tag{2}$$

ここで，Voc は Vocabulary を指し，G の全てのグラフにおける部分グラフの集合である．学習済みモデルでは、類似構造を持つグラフ同士が、ベクトル空間上の近傍にマッピングされる．よって FCSG に Graph2vec を適用することで，類似した機能を持つ FCSG 同士がベクトル空間上で近い位置にマッピングされることが期待できる．

3. 先行研究

この節では，先行研究 [7] の FCSG を用いたマルウェアの機能推定手法について説明する．

3.1 Function Call Sequence Graph

マルウェアの機能はシステムコールや C 言語の標準ライブラリ関数を複数呼び出すことによって実現されるが，これらの呼び出し順序は ISA に依存しない．Function Call Sequence Graph (FCSG) は IDA Pro による静的解析を用いて，マルウェア検体から関数呼び出しを抽出し，その時系列情報を有向グラフ化したものである．つまり，FCSG の各ノードは関数名であり，エッジはある関数から次に呼ばれる可能性のある関数に向けて有向枝がはられる．検体の FCSG は IDA Pro によって検体を逆アセンブルし，関数毎の制御フローを取得した後，制御フローの分岐を考慮しながら実行順にアセンブリコードを取得し，それが関数呼び出し命令であった場合に，その呼び出し先の関数名を取得することで生成される．

3.2 シグネチャ-FCSG を用いた機能推定

先行研究 [7] では，公開されている Bashlite と Mirai のソースコードを目視で確認しながら機能で区切り，区切った部分毎に関数呼び出しを抽出しシグネチャ-FCSG を作成する．Bashlite に対しては 13，Mirai に対しては 15 のシグネチャを作成している．作成されたシグネチャの一覧を表 1，表 2 に示す．これらのシグネチャは，異なる ISA のの命令を含んでおらず，ISA が異なる検体間でも横断的に使用可能であることを特徴としている．機能推定は，シグネチャ-FCSG が検体の FCSG に部分グラフとして含まれるかどうか判定することで，その検体がシグネチャに対応する機能を有しているかどうか判定する．なおシグネチャマッチングには VF2 [1] というグラフマッチングアルゴリズムを用いている．

表 1 Bashlite のシグネチャ一覧

シグネチャ名	機能
access_to_c2	C2 サーバーにアクセス可能かどうかを確認
echo_build_info	C2 サーバーに自身のバイナリ情報を送信
copy_process	プロセスの複製
ping_pong	C2 サーバーから ‘PING’ が送られてきた際に ‘PONG’ と返答する
exec_command	C2 サーバーから送られてきたコマンドを実行する
get_parameter	C2 サーバーから送られてきたパラメータを取得する
get_ip	自端末の IP を取得する
get_mac	自端末の MAC アドレスを取得する
get_random_str	ランダムな文字列を取得する
sendHOLD	HOLD 攻撃を実行する
sendJUNK	JUNK 攻撃を実行する
sendTCP	TCP 攻撃を実行する
sendUDP	UDP 攻撃を実行する

表 2 Mirai のシグネチャ一覧

シグネチャ名	機能
(main) init_argcheck?	コマンドライン引数を確認
(main) init_hide_argv0	コマンドライン引数を隠蔽
(main) init_hide_pname	プロセス名を隠蔽
(main) init_print_system_exec	起動しているプロセスを出力
(main) initializer	各種初期化
(main) init_connect	C2 サーバーへの接続を確立
(scanner) scanner_init_init	他端末スキャナの初期化
(scanner) scanner_init_main	他端末スキャナ実行
(attack) attack_parse	攻撃パラメータの取得
(attack) attack_start	プロセスを生成して攻撃開始
(attack) add_attack	攻撃用関数の登録
(attack_app) attack_app_http	HTTP に対する攻撃
(attack_app) attack_app_cfnull	HTTP に対する攻撃 (large POST)
(attack_gre) attack_gre_ip	GRE IP 攻撃
(attack_gre) attack_gre_eth	GRE Ethernet 攻撃

3.3 系統樹によるマルウェアクラスタリング

マルウェア解析において，クラスタリングは膨大なマルウェア検体を効率良く解析するのに有効な手法の一つである．クラスタリングを行うことで似ている検体をいく

表 3 マルウェアファミリー名と ISA の内訳

ファミリー \ ISA	ARM	MIPS	Interl 80386	x86-64	Power PC	MC68000	SPARC	SuperH	Total
Bashlite	4876	4440	4474	2157	2328	2062	2034	189	22560
Mirai	56	64		1	1		1		123
Tsunami	110	118	94	49	53	44	38	47	553
Lightaudra	2					9			11
SINGLETON	1	3		1	2	4			11
None	902	792	724	369	371	363	345	37	3903
Total	5947	5417	5292	2577	2755	2482	2418	273	27161

つかのクラスタにまとめることができる．本稿では，大量のマルウェアをクラスタリングした後，各クラスタに FCSG を用いた解析が応用できるか検討した．今回は，He ら [2] の系統樹を用いたクラスタリングを行い，生成された各クラスタ内の検体に対して先行研究 [7] でのシグネチャマッチングを行った．このクラスタリング手法では，Lempel-Ziv-Markov chain アルゴリズムを用いて検体のバイナリファイルを圧縮した後, 正規化圧縮距離 (Normalized Compression Distance) を用いて検体間の類似度 (距離) を測り，その距離に基づいてクラスタリングを行う．また，クラスタリングの際には，高速近似法と呼ばれる手法を用いて，精度を保ちつつ計算量を大幅に削減している．

4. 提案手法

この節では，柔軟なシグネチャマッチングを行うための本研究の手法について説明する．

4.1 アプローチ

本研究では，先行研究 [7] と同様の検体の FCSG とシグネチャ-FCSG を作成した後，グラフマッチングを実施していた部分に代わりにグラフ埋め込み手法を適用し，より柔軟にシグネチャマッチングを実施することを目指す．ここで，検体の FCSG とシグネチャ-FCSG のグラフのサイズには大きく差があるため，まず柔軟なマッチングが必要とされる検体の FCSG から該当シグネチャに最も類似していると思われる部分グラフ (以下，類似部分グラフと呼ぶ) を抽出し，その後，Graph2vec によって類似部分グラフとシグネチャ-FCSG をベクトル空間に埋め込み，類似部分グラフとシグネチャ-FCSG の特徴ベクトル間の類似度を算出することにする．

4.2 提案手法の必要性の確認

まず，作成された各シグネチャにおいて柔軟なマッチングが必要と思われる検体が存在するかどうかを確かめることにした．横浜国立大学の吉岡研究室で運用されている IoTPOT [11] で，2016 年 10 月から 2018 年 1 月までに収集された検体のうち，IDA Pro で解析に成功し，main 関数を特定できた検体，すなわち FCSG を作成できた 27,161 検体のうち，各シグネチャにおいて，先行研究 [7] でマッチしなかった検体に対し，それぞれのシグネチャ内に使われている関数の一致数を調査した．表 3 に，27,161 検体の命令セットアーキテクチャと AVClass [9] で取得したマルウェアファミリー名の内訳を示す．ファミリー名が SINGLETON の検体は AVClass でファミリー名を判定できなかった検体である．また，ファミリー名が None の検体は VirusTotal に未アップロードだった検体である．関数の一致数の調査結果を Bashlite のシグネチャに関しては表 4 に，Mirai のシグネチャ関しては表 5 に示す．ここで，表 4 中の 0〜7 と表 2 中の 0〜15 は各シグネチャに対する関数の一致数を示しており，シグネチャ名の横の数字は各シグネチャのノード数，すなわち使用されている関数の数を表している．例えば，調査に使った 27,161 検体のうち，Bashlite のシグネチャ get_random_str に先行研究 [7] で完全マッチしなかった検体が 1,876 検体存在し，このうち get_random_str で用いられている関数と同じ関数を 2 つ持っている検体が 77 存在していることを意味している．つまり，この 77 検体は get_random_str に完全マッチしていないが，使用されている関数を全て持っているということになる．2 つの表から，シグネチャにマッチしていない検体でも，そのシグネチャ内で使われている関数を多く持つ検体を確認することができる．

その中でも表 4 よりシグネチャ exec_command は 7 つの関数で構成されるシグネチャであるが，その全てを持つににかかわらず，先行研究 [7] でマッチさせることができていない検体数が 12,135 検体存在している．このシグネチャは特に柔軟なマッチングが必要とされると検体が多いと考えられ，今回はこのシグネチャのみに焦点を当てて調査を行った．

4.3 類似部分グラフの抽出

検体の FCSG からの類似部分グラフの抽出について説明する．関数の一致数がシグネチャの関数の数と等しい場合，シグネチャに対する類似部分グラフを「シグネチャ内の全ての関数を順番通りに通る，開始関数から終了関数までの最短経路」で定義する．ここで開始関数とはシグネチャの中で最初に呼び出される関数であり，終了関数は

表 4 Bashlite シグネチャにおける完全マッチしなかった検体に対する関数一致数

シグネチャ名 \ 関数一致数	0	1	2	3	4	5	6	7	Total
get_random_str (2)	1152	647	77						1876
access_to_c2 (3)	7	276	1557	90					1930
copy_process (3)	69	17	321	48					455
ping_pong (3)	746	226	444	21					1437
echo_build_info (4)	1083	675	4826	3139	2591				12314
get_parameter (4)	68	0	3	7579	1285				8935
sendHOLD (4)	199	583	944	14869	2				16597
sendJUNK (4)	199	583	940	14922					16644
sendTCP (4)	206	642	261	950	72				2131
sendUDP (4)	206	597	431	1547	72				2853
get_ip (6)	6	57	9	3	458	2748			3302
get_mac (6)	6	0	128	631	2048	558	1		3372
exec_command (7)	60	616	382	786	1914	1160	3295	12135	20348

表 5 Mirai シグネチャにおける完全マッチしなかった検体に対する関数一致数

シグネチャ名 \ 関数一致数	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Total
initializer (2)	26953	33															26986
add_attack (2)	26792	175	20														26987
init_hide_argv0 (3)	26663	118	49	21													26851
init_hide_pname (3)	1775	24920	138	21													26854
attack_parse (3)	26676	307	94	0													27077
init_connect	6	699	26444	5	4												27158
init_print_system_exec (4)	177	26758	0	20	32												26987
init_argcheck? (4)	26731	0	19	234	2												26986
attack_start (5)	68	11	153	279	22509	4032											27052
scanner_init_main (6)	90	1223	25122	622	118	1	0										27156
attack_app_http (7)	26676	44	215	51	42	108	0	0									27136
attack_app_cfnull (8)	1177	1743	23740	149	288	58	3	0	0								26987
scanner_init_init (12)	60	8	1	0	29	1035	25517	155	162	68	120	1	0				26987
attack_gre_ip (13)	5	57	7	340	823	188	1741	23656	320	18	0	3	0	0			27158
attack_gre_eth (15)	5	57	7	4	347	774	286	1634	23694	227	120	0	0	3	0	0	27158

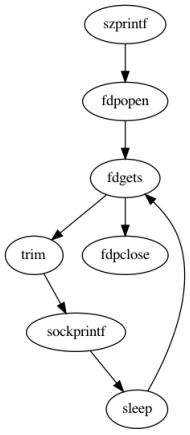

図 1 exec_command の FCSG

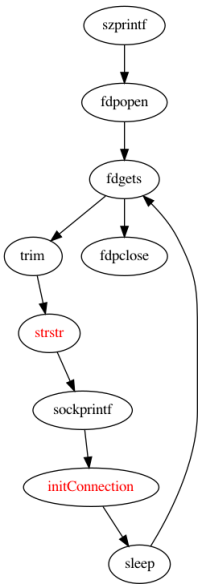

図 2 抽出した類似部分グラフの例

最後に呼び出される関数である．今回焦点を当てているシグネチャ exec_command の FCSG を図 1 に示し，作成された exec_command に対する類似部分グラフの例を図 2 に示す．これはシグネチャ内に strstr，initConnection という 2 つの関数が追加されたことで，図 1 の部分グラフの構造が無くなり，先行研究でマッチしなかったと分かる．このようにして，12,135 検体の FCSG からシグネチャ exec_command に類似していると思われる構造の部分グラフを検体の FCSG から抽出した．

5. 調査結果

この節では，今回提案した手法で確認できた結果について報告する．

5.1 シグネチャ内の関数追加の確認

まず関数一致数が最大の 7 であった 12,135 検体に対して，シグネチャ内に追加されていた関数の組み合わせについて調査した．表 6 はその上位 10 種類を示した表である．ここで最短経路長は，4.3 節で定義した最短経路の長さである．表 6 より最短経路長は最大でも 12 であり，exec_command に非常に類似した構造の部分グラフを持つ検体が多数存在していることが確認でき，追加された関数も多くても 5 個程度ということからも機能面では，大きな違いは無いと考えられる．また，図 2 のような strstr,initConnection という 2 つの関数がシグネチャ内に追加された構造の部分グラフを持つ検体が 10,876 検体存在しており，それが全体の約 9 割を占めていた．また，それ以外に追加されている関数を見てみても，この 2 つの関数のどちらか一方または両方が追加されていることを確認することができる．strstr は文字

表 6　シグネチャ exec_command 内に追加されていた関数の上位 10 種

関数名	最短経路長	検体数
(strstr, initConnection)	9	10876
(initConnection)	8	551
(strstr, strstr)	9	262
(strlen, listFork, initConnection, strcoll, recvLine)	12	80
(send, strlen, initConnection, get_telstate_host)	11	48
(initConnection, strstr, getBulid, exit)	11	47
(initConnection, listFork, strstr, exit)	11	39
(initConnection, listFork, strstr, strtok, recvLine)	12	33
(initConnection, listFork, strstr)	10	21
(initConnection, strstr, getPrefix)	10	20

列を検索するための関数であり，initConnection は C2 サーバへの接続を確立するための関数である．よって関数一致数が最大の 7 であったこれらの検体は元の exec_command の機能にこれらの関数の機能が追加された exec_command に類似した機能を持つ検体が多いと推定することができる．

5.2　Graph2vec による埋め込み実験

続いて，Graph2vec を用いた柔軟なシグネチャマッチング手法の実験結果について報告する．

5.2.1　対象データセット

Graph2vec の学習に用いるグラフのデータセットには，抽出した 12,135 検体の類似部分グラフに加え，その比較対象となるグラフも一定数必要であると考えた．そこで，関数一致数が 4 であった 1,914 検体，5 であった 1,160 検体，6 であった 3,295 検体に対しても部分グラフを抽出することにした．これらの検体はシグネチャ内の関数が少なくとも 1 つは削除されているので，4.3 節のようにシグネチャ内の全ての関数を順番通りに通る，開始関数から終了関数までの最短経路を考えることができない．なので，経路途中で存在しない関数があった場合は，その次に呼び出される関数への最短経路を考えた．つまり，i 番目に呼び出される関数から $i+1$ 番目に呼び出される関数への最短経路を考える際，$i+1$ 番目の関数が存在しなければ，$i+2$ 番目に呼び出される関数への最短経路を，$i+2$ 番目も存在しなければ，$i+3$ 番目というようにして，その検体の FCSG の中で exec_command の中に使われている関数のうち，最も先に呼び出さる関数から最後に呼び出される関数への最短経路途中で通ったノードで構成された部分グラフを抽出した．

関数一致が 3 以下の検体は，exec_command に類似した機能を持つことは考えにくい上に，このような手法で部分グラフを抽出すると，関数一致が多いものに比べ，グラフサイズが小さくなると考えたため，以上の関数一致が 4〜7 の計 18,504 検体に対して抽出した類似部分グラフと exec_command のシグネチャ-FCSG の合計 18,505 個のグラフを Graph2vec のデータセットに用いた．

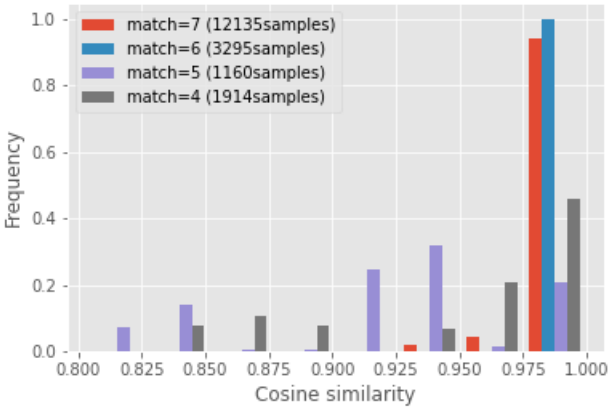

図 3　類似部分グラフとシグネチャ間のコサイン類似度

5.2.2　設定

今回は，ベクトル空間への埋め込み後，Graph2vec によって得られた各検体の類似部分グラフの特徴ベクトルと exec_command のシグネチャ-FCSG の特徴ベクトル間の類似度をコサイン類似度によって測ることにした．n 次元ベクトル $\vec{u} = (u_1, u_2, \cdots u_n)$ と $\vec{v} = (v_1, v_2, \cdots v_n)$ のコサイン類似度 $\cos(\vec{u}, \vec{v})$ は次式で定義される．

$$\cos(\vec{u}, \vec{v}) = \frac{\vec{u} \cdot \vec{v}}{||\vec{u}||||\vec{v}||} \tag{3}$$

コサイン類似度は 2 つのベクトルがどれくらい類似しているかを表し，値が -1 に近ければ類似度が低く，1 に近いほど類似度が高いことを意味している．また，根付き部分グラフの最大深さ D は，4 で実験を行い，Graph2vec で学習した特徴ベクトルの次元数を 128 次元に設定している．

5.2.3　実験結果

図 3 は，各類似部分グラフの特徴ベクトルと exec_command のシグネチャ-FCSG の特徴ベクトル間の類似度をコサイン類似度によって測った結果を関数一致の数毎にヒストグラムで表したものである．match は類似部分グラフ中の exec_command との関数の一致数を表している．なお縦軸は，それぞれの類似部分グラフの検体数が合計 1 になるように正規化してある．図 3 より，関数一致数が 7 であった 12,135 検体の類似部分グラフを柔軟なマッチングが必要な検体という前提を置き，仮に類似度が 0.9 以上であれば，exec_command と類似した機能を有するという閾値を設定すれば，12,135 検体全ての部分グラフが柔軟なマッチングに成功していた．しかし，関数一致数が 6 に対する類似部分グラフも全て柔軟なマッチングにマッチしている上に，関数一致数が 4,5 個の検体に対する類似部分グラフも閾値を超えているものが一定数存在している．よって，これらが柔軟なマッチングをさせる必要があるのか確かめることが今後の課題点と言える．

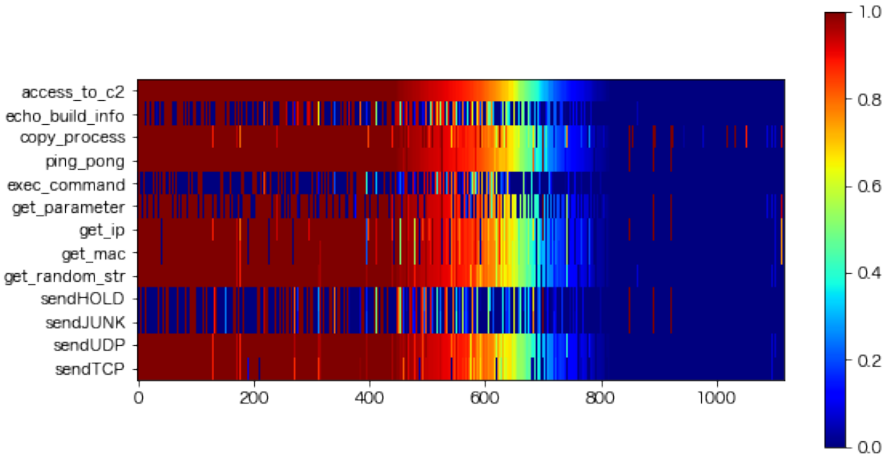

図 4 全クラスタ毎に対する Bashlite シグネチャのマッチ数/検体数のヒートマップ (access_to_c2 でソート)

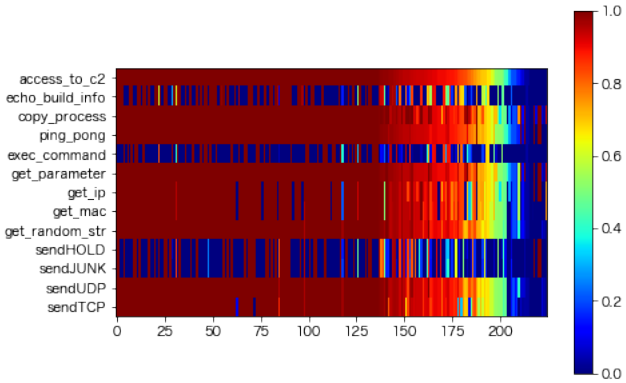

図 5 ARM の Bashlite クラスタ毎に対するシグネチャマッチ数/検体数のヒートマップ (access_to_c2 でソート)

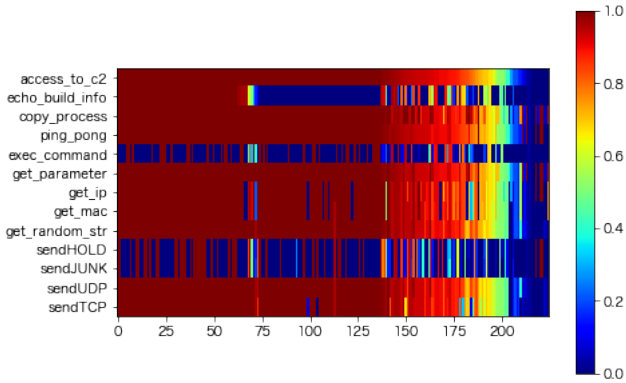

図 6 ARM の Bashlite クラスタ毎に対するシグネチャマッチ数/検体数のヒートマップ (access_to_c2 と echo_build_info でソート)

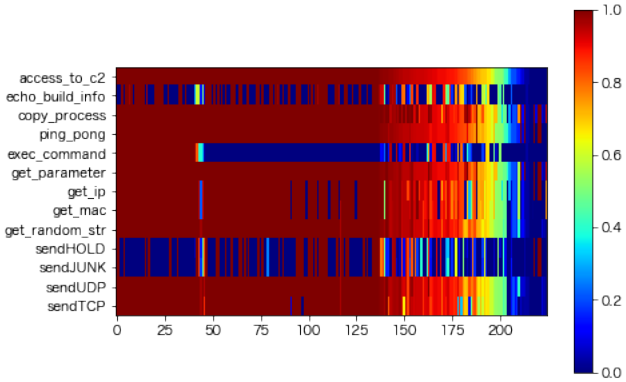

図 7 ARM の Bashlite クラスタ毎に対するシグネチャマッチ数/検体数のヒートマップ (access_to_c2 と exec_command でソート)

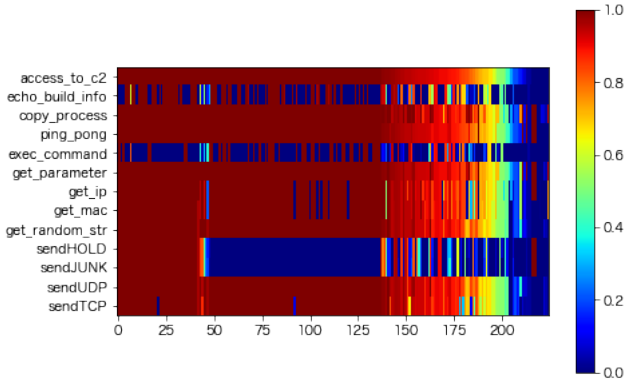

図 8 ARM の Bashlite クラスタ毎に対するシグネチャマッチ数/検体数のヒートマップ (access_to_c2 と sendHOLD でソート)

5.3 クラスタに対する先行研究の応用

続いて，4.2 節の FCSG 作成に成功した表 3 のデータセットに対して，He ら [2] の手法で，検体のクラスタリングを行い，生成された各クラスタ内の検体に対して，先行研究 [7] のシグネチャマッチング手法を適用した結果と考察を示す．

今回のデータセットは，表 3 より Mirai の検体が少ないことから，Bashlite のシグネチャのみを用いてクラスタに対する分析を行った．また，各クラスタにおけるマルウェア科名と ISA は多数決で定めている．

これから示すいくつかのヒートマップの横軸はクラスタを表している．図 4 は，生成された全 1,116 個のクラ

スタに対して，クラスタ毎に (各シグネチャにマッチしたクラスタ内の検体数/クラスタに含まれる検体数) を計算し，access_to_c2 に対するマッチ数で降順にソートしたものである．この図から，クラスタ単位で見ると，シグネチャのマッチ数に傾向があることが確認できる．例えば，access_to_c2，copy_process，ping_pong，get_ip，get_mac，get_random_str，sendUDP，sendTCP の全てにマッチする検体の比率が高く，これらの機能は比較的メジャーな機能であり，これら以外の機能はマイナーな機能であり，攻撃者によって必要ないと判断され機能が削除されたと推定することができる．

また，図 4 のようにクラスタ全体をみると解釈が困難となるので，ファミリー名や ISA が同一であるクラスタに対して追加調査を行った．図 5 は図 4 の中から ARM の Bashlite クラスタ計 225 個を取り出し，同様に access_to_c2 に対するマッチ数で降順にソートさせたものである．図 5 では，ほとんどのクラスタに対して，Bashlite のシグネチャを一定数マッチさせることができていることが確認でき，ある程度機能を考慮したクラスタリングが行われていることを確認した．

さらに，図 6，図 7，図 8 はそれぞれ，access_to_c2 でソートした上に，マッチ数に傾向が見られなかった echo_build_info，exec_command，sendHOLD のマッチ数で降順にソートさせたものである．3 つの図から，これらのシグネチャはお互いに傾向が無いと考えられるが，図 8 から sendHOLD と sendJUNK にはマッチ数に傾向が見られ，これらは互いに関係性の強い機能であると考えられ，攻撃者によって削除される場合も共に削除される可能性が高いと推定することができる．

6. まとめと今後の課題

本稿では，先行研究 [7] でマッチさせることができなかった特定のシグネチャに対して，シグネチャ内に少数の関数が追加されているのみで，機能的にマッチさせる必要性が高い検体の存在と追加される関数にも傾向があることを確認した．また，シグネチャの開始関数から終了関数までの最短経路の情報から，シグネチャに類似した構造の部分グラフが抽出できることを示し，それにグラフ埋め込み手法を用いた柔軟なシグネチャマッチング手法について検討した．加えて，クラスタリングの結果と先行研究 [7] のシグネチャマッチングを横断的に分析することで，クラスタ内の検体における各シグネチャのマッチ数に傾向があることを確認した．

今後の課題としては，各シグネチャに対して，柔軟なシグネチャマッチが必要かどうかの確認が挙げられる．今回は関数の一致数を調べたが，検体のソースコードなどから必要性を特定するなどの方法を検討する予定である．また，strip 処理されシンボル情報が消えた検体に関しては，main 関数やその他の関数名が特定できず，FCSG を作成できない．このような strip された検体に対しては IDA F.L.I.R.T. [3] と呼ばれる関数の先頭バイト列をシグネチャとした関数特定技術を用いることで，main 関数を推定する手法を検討している．

謝辞 本研究は総務省の「電波資源拡大のための研究開発 (JPJ000254)」における委託研究「電波の有効利用のための IoT マルウェア無害化／無機能化技術等に関する研究開発」によって実施した成果を含む．

参考文献

[1] Cordella, L. P., Foggia, P., Sansone, C. and Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 26, No. 10, pp. 1367–1372 (2004).

[2] He, T., Han, C., Isawa, R., Takahashi, T., Kijima, S., Takeuchi, J. and Nakao, K.: A Fast Algorithm for Constructing Phylogenetic Trees with Application to IoT Malware Clustering, *Neural Information Processing* (Gedeon, T., Wong, K. W. and Lee, M., eds.), Cham, Springer International Publishing, pp. 766–778 (2019).

[3] Hex-Rays SA: Ida f.l.i.r.t. technology: In-depth - hex rays, `https://hex-rays.com/products/ida/tech/flirt/in_depth/`. (Accessed on 08/19/2021).

[4] Hex-Rays SA: IDA Pro – Hex Rays, `https://www.hex-rays.com/products/ida/`. (Accessed on 08/14/2021).

[5] ifding: GitHub - ifding/iot-malware: Malware source code samples leaked online uploaded to GitHub for those who want to analyze the code, `https://github.com/ifding/iot-malware/`. (Accessed on 08/14/2021).

[6] jgamblin: GitHub - jgamblin/Mirai-Source-Code: Leaked Mirai Source Code for Research/IoC Development Purposes, `https://github.com/jgamblin/Mirai-Source-Code`. (Accessed on 08/14/2021).

[7] Kawasoe, R., Han, C., Isawa, R., Takahashi, T. and Takeuchi, J.: Investigating Behavioral Differences between IoT Malware via Function Call Sequence Graphs, *Proceedings of the 36th ACM/SIGAPP Symposium on Applied Computing (SAC)* (2020).

[8] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S. and Dean, J.: Distributed representations of words and phrases and their compositionality, *Advances in neural information processing systems*, pp. 3111–3119 (2013).

[9] Monrose, F., Dacier, M., Blanc, G. and Garcia-Alfaro, J.(eds.): *AVclass: A Tool for Massive Malware Labeling*, Cham, Springer International Publishing (2016).

[10] Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y. and Jaiswal, S.: graph2vec: Learning distributed representations of graphs, *arXiv preprint arXiv:1707.05005* (2017).

[11] Pa, Y. M. P., Suzuki, S., Yoshioka, K., Matsumoto, T., Kasama, T. and Rossow, C.: IoTPOT: analysing the rise of IoT compromises, *9th USENIX Workshop on Offensive Technologies (WOOT 15)* (2015).

[12] Shervashidze, N., Schweitzer, P., Van Leeuwen, E. J., Mehlhorn, K. and Borgwardt, K. M.: Weisfeiler-Lehman graph kernels., *Journal of Machine Learning Research*, Vol. 12, No. 9 (2011).