Computer Security Symposium 2020
26 - 29 October 2020

関数呼び出しシーケンスに着目した IoTマルウェアの機能差分調査

川添 玲雄[1,a)] 韓 燦洙[1,2] 伊沢 亮一[2] 高橋 健志[2] 竹内 純一[1]

概要：IoT 機器に感染するマルウェア（IoT マルウェア）が猛威を奮っている．これらの多くは Mirai や Bashlite などの公開されているソースコードをもとに，機能の追加や変更，削除をすることで作成されている．そのためアンチウイルスソフトなどでマルウェアの科名を得るだけでは，亜種間の機能の差分を捉えることができない．本研究では，単に亜種の流行を確認するだけでなく，機能差分を考慮した上で亜種の実態を調査する目的で，シグネチャを用いたマルウェアの機能判定手法を提案する．ここでシグネチャとはマルウェアが有する各機能に対応する関数のコールシーケンスグラフ（CSG）であり，解析者が半手動で作成する．このシグネチャが対象のマルウェア検体の CSG に対して部分グラフとしてマッチすると，対応する機能を有するものとする．本シグネチャは異なる CPU アーキテクチャ間の亜種に対しても横断的に使用できることを特徴とする．本稿では，24,126 検体に対して機能判定を実施し，複数ファミリーの機能を合わせもった混合亜種の存在や，一部の機能に特化した検体の存在をケーススタディとして示す．

キーワード：IoT マルウェア，マルウェア解析，静的解析

A Feasibility Study on Investigating Functional Differences between IoT Malware

Reo Kawasoe[1,a)] Chansu Han[1,2] Ryoichi Isawa[2] Takeshi Takahashi[2] Jun'ichi Takeuchi[1]

Abstract: IoT malware is created by editing functions based on publicly available source codes. Therefore, it is not possible to capture the functional differences between variants by simply obtaining the malware family name with antivirus software. In this research, we propose a method of malware function determination using signatures for the purpose of confirming the epidemic of subspecies and investigating the actual status of subspecies in consideration of functional differences. The signature is a call sequence graph (CSG) corresponding to each function and is created manually by an analyst. If this signature matches the CSG of the target sample as a subgraph, it has the corresponding function. This signature can be used across variants of different CPU architectures. In this paper, we performed function determination on 24,126 specimens, and showed the existence of mixed subspecies with functions of multiple families and specimens specialized for some functions as case studies.

Keywords: IoT malware, Malware analysis, Static analysis

1. はじめに

IoT 機器の普及に伴い，脆弱な機器に感染する IoT マルウェアも増加しており，2020 年の第 1 四半期には 70,000 以上もの新しい IoT マルウェアが確認 [10] されている．IoT マルウェアには，それぞれに固有の感染手法や攻撃手法

[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)] kawasoe@me.inf.kyushu-u.ac.jp

© 2020 Information Processing Society of Japan

を持つ複数のファミリーが存在する．そのうち，Bashlite や Mirai などのいくつかのファミリーにおいては，ソースコードがインターネット上に公開されている [1], [7]．そのため，第三者がそれらソースコードを入手し，それに対して機能の追加や変更，削除したマルウェア，いわゆる亜種を作成可能である．

インターネット上の IoT 機器に感染を広げる IoT マルウェアを捕獲するハニーポット [9], [11] が運用されている．吉岡ら [11] が運用するハニーポットが 2016 年 10 月から 2018 年 1 月の期間に捕獲したマルウェアのうち約 2 万検体に対し，アンチウイルススキャナーでマルウェア科名を調べると Bashlite や Mirai などの 6 種類（Unknown 含む）のみのマルウェア科名が得られた．しかしながら，同じマルウェア科名をもつ検体間でも機能の差異はあると考えられ，マルウェア科名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，など機能差分に着目した亜種の実態を知ることは難しい．

本稿では機能差分を考慮した上での亜種の実態を調査することを目的に，マルウェア検体間の機能差分を求める手法を提案する．提案手法はマルウェアの機能に対応するシグネチャをもとに，検体に対してシグネチャマッチングを実施，マッチしたシグネチャの機能を有するものと判定する．ここでシグネチャとは機能に対応する複数の関数のグラフであり，関数の呼び出し順を保持した有向グラフ（以下，CSG（Call Sequence Graph）と呼ぶ）である．シグネチャとなる CSG は解析者がマルウェアのソースコードや逆アセンブルコードから機能とそれに 関連する関数を半自動で抽出して作成する．本シグネチャは異なる CPU アーキテクチャの命令（例えば Intel の jmp 命令と ARM の BL 命令）を含んでおらず，CPU アーキテクチャが異なる IoT 検体間でも横断的に使用可能であることを特徴とする．機能判定の際には，対象検体のバイナリから CSG を作成し，その CSG のサブグラフとしてシグネチャがマッチするか検査する．

本稿では吉岡らのハニーポットで捕獲した約 2 万検体に対して提案手法を適用することで機能差分調査を実施し，以下の知見が得られた．

- マルウェア科名としては 6 種類であるが，あるマルウェア科名のソースコードに対して別のマルウェア科名のソースコードを流用して作成された混合亜種を確認
- 一部機能が削除され，特定の機能に特化した可能性がある検体を確認
- Github 公開されているソースコード全てもしくは大部分をそのまま使用して作成された検体を確認．ソースコードをほぼ改変しないマルウェア作者も存在する．

本稿の構成は以下の通りである．第 2 節で基礎知識を説明した後，第 3 節で提案手法を説明する．第 4 節では提案手法を実際に適用した結果を示し，第 6 節では本手法に関連する研究について述べ，第 7 節でまとめを述べる．

2. 基礎知識

本節ではマルウェア，特に IoT マルウェアを分析する際に必要な要素について述べる．

2.1 命令セットアーキテクチャ

IoT 機器には Linux が組み込まれている事が多いが，用いられている CPU には様々なものが存在する．そのため，IoT 機器を狙ったマルウェアは，様々な CPU，つまり命令セットアーキテクチャ（ISA）上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．一方で，同じソースコードから生成した場合でも，ISA が異なると，その ISA で動作するバイナリの構造が異なる．よって，ISA の影響を受けず，単純にそのバイナリが持つ機能だけに着目することが可能となれば，解析作業の軽減が期待できる．

2.2 逆アセンブル

一般的にプログラムは，人間が理解可能な高級言語でソースコードとして記述された後，コンパイラによって低級言語のバイナリファイルへと変換される．マルウェアを分析する場合，一部の例を除いて，ソースコードが入手できない場合がほとんどである．バイナリファイルのみが利用可能な場合，手法としては表層解析，動的解析，静的解析のいずれかを取る必要がある．

逆アセンブルは静的解析の手法の一つであり，機械語の羅列であるバイナリファイルをアセンブリ言語へと変換する処理のことである．IDA Pro[6] は Hex Rays が開発している逆アセンブルツールであり，バイナリファイルを解析してアセンブリ言語へと変換する．更にジャンプ命令を認識して，アセンブリ言語の制御フローを生成する機能を有する．

2.3 コールシーケンスグラフ

CSG（Call Sequence Graph）は有向グラフの形で表される． CSG の各ノードは関数名であり，エッジは「ある関数名」から「一連の処理の中で，ある関数が呼ばれた後，その次に呼ばれる可能性がある関数名」へと引かれる．

関数の呼び出し関係を示すグラフに，似たものとして「コールグラフ」というものが存在する．コールグラフは関数の呼び出し，被呼び出し関係を表現できるグラフであるが，関数を呼び出した時系列の情報は失われてしまう．これに対して CSG は関数が呼び出される時系列の順序が表現可能であるため，より機能に応じたグラフが生成されることが期待できる．

菊地ら [15] は「API コールグラフ」を定義しているが，提案手法ではシステムコール等に限らず，すべての関数

呼び出しをグラフに含めるため，CSG と呼ぶようにしている．

2.4 システム関数

Linux のシステムコールや，C 言語の標準ライブラリ関数は，関数が有する機能が既に明らかである．それらの関数を「システム関数」と呼ぶことにする．本稿では JM Project が提供している man ページ [8] に含まれる関数を，システム関数としている．

3. 提案手法

本節ではシグネチャマッチングによる機能判定手法を提案する．

3.1 基本アイデア

異なる CPU アーキテクチャの IoT マルウェア検体に対して横断的にシグネチャマッチングするため，提案手法では機能に対応する関数の呼び出し順序を考慮した有向グラフ（CSG: Call Sequence Graph）とノードとなる関数名をシグネチャとして使用する．これはマルウェアの機能は複数の関数により実装されているという考えに基づいている．

CSG に対して，仮に IoT 検体のバイナリからシグネチャを作成することを考えると，各 CPU アーキテクチャの命令セットの差異（例えば，Intel の jmp 命令と ARM の BL 命令）により，横断的にシグネチャマッチングすることが難しくなる．バイナリレベルよりも抽象度の高い CSG を採用することでこの課題を解決する．提案手法のシグネチャとなる CSG は解析者がマルウェア検体のソースコードもしくは逆アセンブルコードより半自動で作成する．

提案手法では対象の IoT 検体のバイナリを入力とし，以下のように自動で CSG を抽出する．入力となるバイナリを逆アセンブルしたあと，main 関数を起点に，コードを静的に解釈して呼び出される関数の CSG を作成する．main 関数から呼び出される関数があり，その関数から呼び出される関数も CSG に追加するといった，再帰的に関数の呼び出しを追う．これは静的解析に属する方法である．一方，CSG の抽出方法として，検体を実行し関数の呼び出しを追う動的解析も考えられるが，トリガーの問題が生じる．特に C&C（Command and Control）サーバから命令を受信しないと呼び出されない関数があり，動的解析ではこれを追うことが容易ではない．そのため提案手法では静的解析を採用している．

シグネチャとなる CSG が検体の CSG にサブグラフとして含まれるのであればその機能を有するものとして判定する．単にグラフの形状だけでマッチングさせると，誤判定が多くなるという考えのもと，マッチングにはノードとなる関数名のマッチングも実施している．このシグネチャマッチングは既存のグラフマッチングアルゴリズムにて実現可能である．

3.2 機能に対応するシグネチャ CSG の作成

公開されているソースコードや解析済みの逆アセンブル結果を目視確認し，機能で区切る．その後，区切った部分毎に関数呼び出しを抽出，関数名に対する処理（後述）を適用し，CSG を作成する．作成された CSG を「機能と対応したシグネチャ」とする．

本稿では BASHLITE と Mirai のソースコード [1], [7] を基に，BASHLITE に対しては 13，Mirai に対しては 15 のシグネチャを作成した．作成したシグネチャの一覧を**表 1**，**表 2** に示す．

3.3 シグネチャマッチングによる機能判定

CSG 抽出: 初めに，検体を IDA Pro[6] を用いて逆アセンブルを行い，関数毎の制御フローを取得する．次いで，検体内の関数のうち，システム関数でない関数に対して，制御フローの実行順序を維持したまま，関数名を用いて行われている関数呼び出し命令を抽出する．その後，関数名に対しての処理を行い，main 関数の CSG を作成する．

main 関数の CSG のノードのうち，システム関数でないノードについては，該当関数の CSG を同様に作成し，main 関数の CSG に結合する．このような CSG の結合操作を再帰的に繰り返し，最終的に，プログラム実行時に関数名を指定して呼ばれる可能性がある関数全てが含まれる CSG を作成する．

ポインタを用いた関数呼び出しなど，関数名を用いないで呼び出されている関数にも対応できるように，上の作業にて結合されなかった関数についても個々の CSG を作成し，シグネチャマッチングの対象とする．

関数名の処理: 一部のコンパイラは，標準ライブラリ関数に特定の接頭語を付加した上でバイナリを生成する．そのため，コンパイラごとの差異を吸収できるように，関数名の先頭の一部を削除する処理を行う．本稿ではアンダーバーで挟まれた `GI`，`libc`，`uClibc` が存在した場合にアンダーバーを含む該当部分を削除している．

ARM 系の CPU は機械語レベルの mod 演算をサポートしておらず，ソースコード中に mod 演算が存在する場合，同様の演算結果を得られる `__modsi3` 関数へと置き換えられる．また，配列の初期化を行う際に，`memset` 関数が用いられる場合とそうでない場合がある．これらのケースに備えて，`__modsi3` 関数と `memset` 関数においてはその全てを削除する．

グラフマッチング: VF2 と呼ばれるグラフマッチングアルゴリズム [3] を用いて，検体の CSG の中にシグネチャの CSG が埋め込まれているかを探索する．

VF2 アルゴリズムは初期状態で，グラフの形状のみを確認するアルゴリズムであるが，Feasibility rule を定義す

表 1　Bashlite のシグネチャ一覧
Table 1　List of Bashlite signature

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
Table 2　List of Mirai signature

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

ることでノードやエッジに対する一致も確認することが可能である．本稿では「ノード名が一致しているかの確認」を定義した．なお，その際には以下の処理により，確認を行う．

- ノード名を小文字に統一する
- 「文字の出力」に使われる `printf()`，`puts()`，`putchar()` は同一の関数として扱う

実際に検体から抽出した CSG の例を図 1 に示す．なお，シグネチャがマッチした部分は赤で，次に抜き出している部分はピンクで示している．また，図 1 のうち，シグネチャにマッチした部分の一部を 2 と図 3 に示す．

4. 機能差分調査

本節では今回実施した調査について述べる．

4.1 調査対象検体データセット

調査対象のデータセットとして，横浜国立大学の吉岡研究室にて運用されている IoTPOT[11] で 2016 年 10 月 2 日から 2018 年 1 月 20 日までに収集された 54,544 件の検体のうち，IDA Pro による逆アセンブルを行い，`main` 関数が特定できた 24,126 件を用いた．

表 3 に，Linux の readelf コマンドで取得した命令セットアーキテクチャと，AVClass[13] で取得したマルウェアファミリー名の内訳を示す．なお，ファミリー名が NaN の検体はデータ取得時点で VirusTotal に未アップロードだった検体であり，データセットの都合上，アップロードが難しいためそのままにしてある．また，SINGLETON は AVClass でファミリー名判定ができなかった検体である．

4.2 Bashlite のシグネチャによる調査結果

表 4 に Bashlite のシグネチャマッチ数とファミリーの内訳を示す．解析対象のうち，AVClass で Bashlite と判定されているものは 20,026 件であるが，そのうち 19,954 件については少なくとも 1 件以上シグネチャをマッチさせることが可能であった．

AVClass では Bashlite と判定されていたが，1 つも Bashlite のシグネチャがマッチしていない検体が 72 件存在した．それらの検体について調査したところ，今回シグネチャを作成する際に参照したソースコード [7] とは大幅に異なるプログラム構造が確認できた．よって，これらの検体は，マルウェアファミリーという大きな括りでは Bashlite であるものの，公開ソースコード [7] とは異なる変化をしてきた検体であると考えられる．そのため，関数呼び出しの順序が大きく変わってしまい，その結果，シグネチャがマッチしなかったものと考えられる．

一方でシグネチャが一部のみマッチしている検体も存在した．それらの検体を確認したところ，

- get_parameter
- exec_command
- get_ip
- get_mac
- sendHOLD
- sendJUNK

といったシグネチャがマッチしなくなっており，該当する機能が削除されている事が確認できた．このことより，Bashlite においては一部の機能が削除され単純化した亜種が存在すること，そのような検体に対して提案手法で削除された機能を推定できることが確認できた．

4.3 Mirai のシグネチャによる調査結果

表 5 に Mirai のシグネチャマッチ数とファミリーの内訳を示す．Mirai においては，解析対象の検体のうち，AV-

表 3　マルウェアファミリー名と ISA の内訳

Table 3　Breakdown of malware family and ISA in the dataset

ファミリー \ ISA	x64	ARM	x86	MC68000	MIPS	Sparc	Total
Bashlite	2154	4871	4468	2069	4431	2033	20026
Lightaidra		1		3			4
Mirai	1	56			64	1	122
SINGLETON	1			1	3		5
Tsunami	49	110	94	44	118	38	453
NaN	372	909	729	365	795	346	3516
Total	2577	5947	5291	2482	5411	2418	24126

表 4　Bashlite のシグネチャマッチ数とファミリーの内訳（Hit: マッチしたシグネチャ数，Total: 各 Hit に該当するサンプル数）

Table 4　Breakdown of malware family and hit count of Bashlite Signature (Hit: the number of matched signatures, Total: the number of samples for each Hit)

Hit	Bash-lite	Light-aidra	Mirai	SING-LETON	Tsu-nami	NaN	Total
0	72		121		10	70	273
1	130		1		263	68	462
2	642				94	90	826
3	313			1	10	41	365
4	219			2	36	68	325
5	588			2	40	127	757
6	700	2				272	974
7	1487	2				489	1978
8	2944					564	3508
9	3297					521	3818
10	4654					625	5279
11	4799					550	5349
12	178					31	209
13	3						3

表 5　Mirai のシグネチャマッチ数とファミリーの内訳（Hit: マッチしたシグネチャ数，Total: 各 Hit に該当するサンプル数）

Table 5　Breakdown of malware family and hit count of Mirai Signature (Hit: the number of matched signatures, Total: the number of samples for each Hit)

Hit	Bash-lite	Light-aidra	Mirai	SING-LETON	Tsu-nami	NaN	Total
0	19907	4	1	2	453	3404	23771
1	3		52			24	79
2	116			3		58	177
3			2			1	3
4			13			21	34
5			1			1	2
7			50			6	56
9			1				1
10			1			1	2
11			1				1

Class で Mirai と判定されているものは 122 件であり，1 件を除く 121 件に対しては 1 つ以上のシグネチャがマッチ可能であった．

AVClass では Mirai と判定されていたが，Mirai のシグネチャが 1 つもマッチしていない，1 件の検体を調査した．その結果，Mirai だけでなく Bashlite に見られるような関数も存在しており，参照したソースコード [1] とは異なる構造であった．そのため，偶然アンチウイルスソフトが Mirai の構造部分を検出し，AVClass では Mirai と判定されていたものの，シグネチャは 1 件もマッチしなかったものと思われる．

また，Bashlite と比較して Mirai の方がプログラムの構造が複雑であるため，コンパイラによる影響を受ける可能性や IDA Pro によって解析できない関数が存在する可能性が高くなっている．加えて，Mirai は Bashlite よりも今回作成したシグネチャの CSG のサイズが大きい．これらの要因により，Mirai のシグネチャのマッチ数が Bashlite と比べ，低くなっているものと考えられる．

4.4　混合亜種の発見

表 4 と表 5 より，一部の検体については，AVClass にて判定されたファミリー名とは異なるファミリーのシグネチャがマッチしていることが確認できた．これらはある検体に対して異なるマルウェア科名の検体の機能を追加して作成された混合亜種である．

Bashlite の機能を有する Mirai 検体: 表 4 から分かるように，Bashlite のシグネチャがマッチした Mirai の検体が 1 件存在した．この検体に対しては，Bashlite のシグネチャのうち ‘copy_process’ というシグネチャがマッチしていた．一方で，他の Mirai には該当シグネチャがマッチしていない事から，この 1 件は Mirai の亜種であり，Bashlite のソースコードを参考に，プロセスの複製の処理が加えられた可能性が考えられる．

Mirai の機能を有する Bashlite 検体: 表 5 から分かるように，Mirai のシグネチャがマッチした Bashlite の検体が 119 件存在した．これらの検体は Mirai のシグネチャのうち，‘(main) init_hide_argv0’，‘(main) init_hide_pname’

の2つ，あるいは '(main) init_hide_argv0' のみがマッチしていた．このことから，これらの Bashlite の検体は，Mirai のソースコードをコピーすることによってプロセス名，引数の秘匿処理を実装している可能性が考えられる．

5. 制約と今後の課題

本節では提案手法における制約と今後の課題について述べる．

5.1 提案手法における制約

提案手法は，逆アセンブルツールの結果から関数呼び出しシーケンスを抽出し，その関数名を基にグラフマッチングを行い，検体に含まれるシグネチャを検出手法である．そのため，解析結果が逆アセンブルツールの性能に大きく左右されてしまう．加えて，ISA の違いやコンパイラの影響を軽減するために 3.3 節で述べた関数名の処理を行っている．しかしながら，存在する全てのコンパイラの仕様を網羅して対策を取るのは難しい．これらは逆アセンブルを利用している以上，避けられない問題であるため，いかに影響を抑えるかが重要となる．

5.2 今後の課題

提案手法のシグネチャには関数名が含まれている．よって，関数名が削除されている検体バイナリにはシグネチャが適用できないため，今回は調査対象のデータセットから除外している．この関数の削除は Linux の strip コマンドをバイナリに使用することで可能となる．今後の課題として，関数名の代わりに関数内部の構造を使うなどして，strip されたバイナリも調査できるようにしたい．

加えて，He ら [5] などのクラスタリング手法と提案手法を組み合わせることで，同一クラスタに含まれる検体間の機能的な差異を得られるかを検証したい．

6. 関連研究

本節では関連研究について紹介する．

6.1 マルウェアのシグネチャに関する研究

Feng ら [4] は，静的解析によって得られた制御フローグラフをベクトル化し，脆弱性を含む部分も同様にベクトル化してシグネチャとすることで，シグネチャとプログラムの類似度を高速に計算する手法を提案している．

Xu ら [14] は，Feng ら [4] の手法を基に，ニューラルネットワークを用いて処理を行うことによって，更に処理速度を向上させている．

Sathyanarayan ら [12] は，静的解析で重要な API コールを抽出し，その出現頻度を計算することで，マルウェアのファミリーに縛られない，マルウェア全体を検出できるシグネチャの作成手法を提案している．

オープンソースで開発されているアンチウイルスソフトの ClamAV では，ファイルのバイト列やファイルハッシュや PE セクションのハッシュをシグネチャとしている [2]．

本研究においては，マルウェアの実行ファイルにマッチするシグネチャではなく，マルウェア内部の機能毎にマッチするシグネチャを考える必要性があった．また，命令セットアーキテクチャを横断した探索を行いたいという要求が存在した．そのため，機能ごとに特徴が異なり，かつ命令セットアーキテクチャの影響を受けにくい「関数呼び出し」に着目してシグネチャを作成する手法を提案した．

6.2 関数呼び出しシーケンスに着目した研究

菊地ら [15] は，Windows マルウェアに対し，静的解析を行って API コールを抽出，有向グラフ化した上でグラフの類似度を計算し，クラスタリングを行う手法を提案している．

筆者ら [16] は，菊地ら [15] の手法を拡張し，API コールに限らずユーザー定義関数までグラフに含められるようにした上で，IoT マルウェアの特徴をグラフで抽出する手法を提案した．本稿では，筆者ら [16] の提案手法を更に拡張した上で，シグネチャマッチングを行う手法を提案した．

7. まとめ

提案手法では，検体から自動的に生成可能なコールシーケンスグラフ（CSG）に，手動で作成したシグネチャとなる CSG が埋め込まれているかを確認することで，マルウェアが特定の機能を有しているかを判定した．各 CSG を作成する際には，特定の処理を行うことで，コンパイラや命令セットアーキテクチャの差異を吸収可能にした．また，提案手法を実データセットに対して適用し，IDA Pro で `main` 関数が特定できた検体のほぼ全てに対して，1 つ以上のシグネチャがマッチ可能であることを確認した．加えて，複数ファミリーの機能を併せ持つ検体の確認や，一部の機能が削除され特定の目的に特化した検体が存在することを確認できた．

謝辞　データセットをご提供くださった横浜国立大学の IoTPOT チームに深く感謝いたします．本研究成果は，JSPS 科研費 JP18H03291，および国立研究開発法人情報通信研究機構の委託研究の支援により得られたものです．

参考文献

[1] anthonygtellez: GitHub - anthonygtellez/BASHLITE: An archive of BASHLITE source code, https://github.com/anthonygtellez/BASHLITE. (Accessed on 12/11/2019).

[2] Cisco and/or its affiliates: ClamavNet, https://www.clamav.net/documents/creating-signatures-for-clamav#creating-signatures-for-clamav. (Accessed on 08/12/2020).

[3] 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).

[4] Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B. and Yin, H.: Scalable Graph-Based Bug Search for Firmware Images, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, New York, NY, USA, Association for Computing Machinery, p. 480–491 (online), DOI: 10.1145/2976749.2978370 (2016).

[5] 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).

[6] Hex-Rays SA: IDA Pro – Hex Rays, `https://www.hex-rays.com/products/ida/`. (Accessed on 08/12/2020).

[7] jgamblin: GitHub - jgamblin/Mirai-Source-Code: Leaked Mirai Source Code for Research/IoC Development Purposes, `https://github.com/jgamblin/Mirai-Source-Code`. (Accessed on 12/11/2019).

[8] JM Project: JM インデックス (LDP manpages) (Japanese), `https://linuxjm.osdn.jp/INDEX/ldp.html`. (Accessed on 12/11/2019).

[9] Luo, T., Xu, Z., Jin, X., Jia, Y. and Ouyang, X.: IoTCandyJar: Towards an Intelligent-Interaction Honeypot for IoT Devices, *BlackHat USA 2017* (2017).

[10] McAfee Labs: McAfee Labs COVID-19 Threats Report, July 2020, `https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-july-2020.pdf`. (Accessed on 08/13/2020).

[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] Sathyanarayan, V. S., Kohli, P. and Bruhadeshwar, B.: Signature Generation and Detection of Malware Families, *Information Security and Privacy* (Mu, Y., Susilo, W. and Seberry, J., eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 336–349 (2008).

[13] Sebastián, M., Rivera, R., Kotzias, P. and Caballero, J.: AVclass: A Tool for Massive Malware Labeling, *Research in Attacks, Intrusions, and Defenses* (Monrose, F., Dacier, M., Blanc, G. and Garcia-Alfaro, J., eds.), Cham, Springer International Publishing, pp. 230–253 (2016).

[14] Xu, X., Liu, C., Feng, Q., Yin, H., Song, L. and Song, D.: Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, (online), DOI: 10.1145/3133956.3134018 (2017).

[15] 菊地寿史，大久保隆夫：共通 API コールグラフによるマルウェア分類手法に関する研究 (2019).

[16] 川添玲雄，韓　燦洙，伊沢亮一，高橋健志，竹内純一：関数呼び出しシーケンスに着目した IoT マルウェアの機能推定に関する考察 (2020).

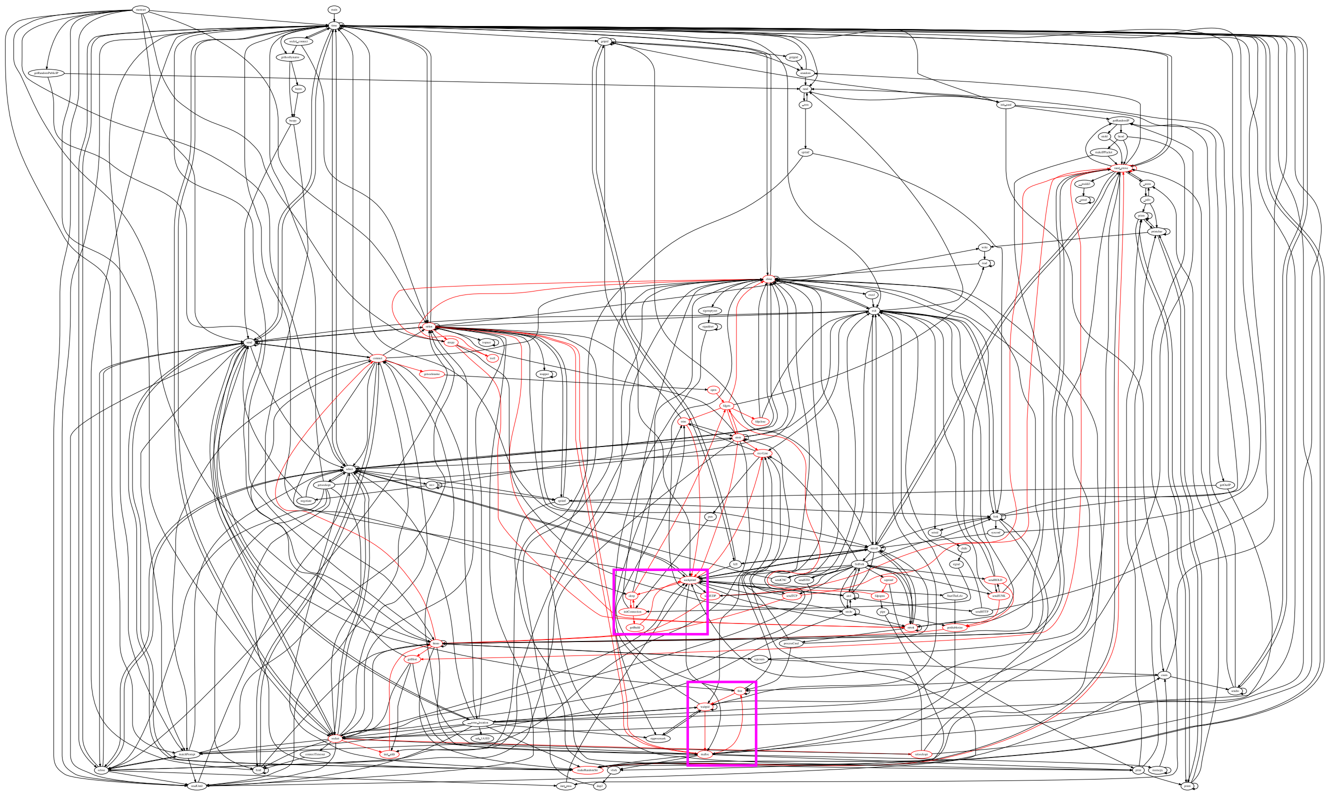

図 1　BASHLITE の検体から抽出した CSG

Fig. 1　Extracted CG from BASHLITE malware

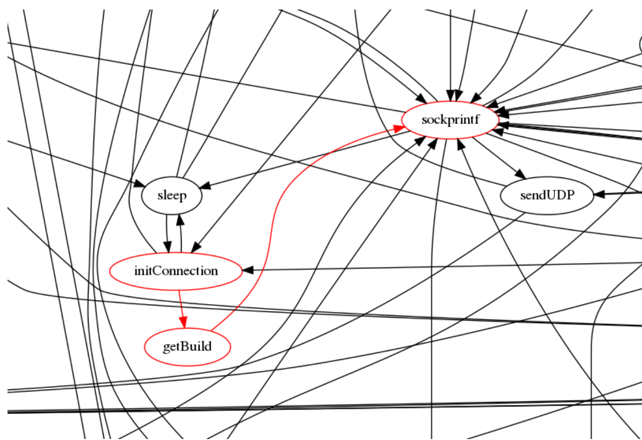

図 2　シグネチャがマッチした CSG の一部 (1)

Fig. 2　Extracted CG from BASHLITE malware

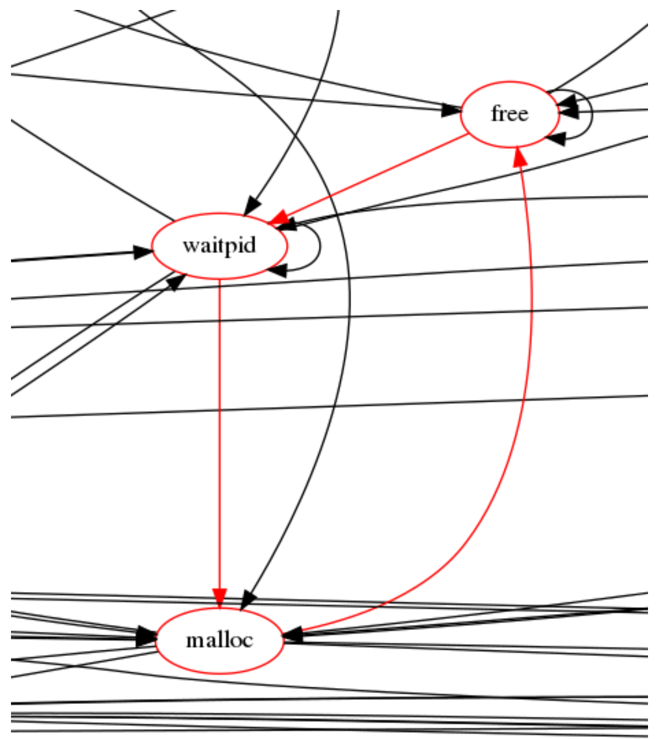

図 3　シグネチャがマッチした CSG の一部 (2)

Fig. 3　Extracted CG from BASHLITE malware