Vol.2018-DPS-174 No.31
Vol.2018-CSEC-80 No.31
2018/3/6

プロセッサレベルの特徴量に注目した亜種マルウェアの検知

大谷 元輝[1]　高瀬 誉[1]　小林 良太郎[2]　加藤 雅彦[3]

概要：近年，IoT 機器の普及によって日常生活の利便性が向上している一方で，IoT 機器を狙ったマルウェアも発生しており，IoT 機器のセキュリティ対策が求められている．しかし，IoT 機器はハードウェア的なリソースが制限されており，PC と同等のマルウェア検知機構を実装することが難しい．そこで，マルウェアの検知機構を外部にオフロードすることでリソースを消費せずにマルウェアの検知を可能とすることを目指す．本研究では，仮想マシンを用いたエミュレーション環境を用意し，オフロードが容易なプロセッサ情報を用いた，マルウェア検知機構を評価した．実験の結果，機能拡張や難読化といった亜種マルウェアの検知が可能であることを確認した．

Detection of Subspecific Malware Focusing on Processor Level Features

Otani Genki[1]　Takase Hayate[1]　Kobayashi Ryotaro[2]　Kato Masahiko[3]

1. はじめに

近年，あらゆるものがネットワークを通じて相互に繋がる IoT(Internet of Things) が発達してきている．IoT 機器は急速に増え続け，2020 年には 304 億個にも上る見通しである [1]．この IoT 機器の普及によって，様々なものの情報をデータ化することができ，集めたデータを元にした産業の自動化や，サービスの新たな付加価値に繋がることが期待されている．一方で，IoT 機器のセキュリティについては未だ課題が多く残されており，現存する IoT 機器の 70%には脆弱性が残されているといわれている [2]．

IoT 機器に感染するマルウェアが引き起こしたセキュリティインシデントも既に発生している．2016 年には，IoT 機器に感染するマルウェア「Mirai」によって構築されたボットネットによる大規模な DDoS 攻撃によって，米 DNS プロバイダ Dyn が大きな被害を受けた [3]．この攻撃により Amazon や Twitter に代表される多くのサービスが一時的に利用不可能となった．このような被害を増やさないためにも，IoT 機器のセキュリティ対策は急務となっている．

また昨今では，亜種マルウェアの開発が盛んに行われている．亜種マルウェアとは，既存のマルウェアのソースコードを一部改変，もしくは機能拡張して作成されたマルウェアを指す．また，バイナリファイルを暗号化して，形を変化させることで静的解析を妨害したり，アンチウイルスエンジンの検知を回避する手法を施されたものも亜種マルウェアとする．本稿では，この手法のことを難読化と定義する．特に IoT 機器に感染するマルウェアは「Mirai」のソースコードが公開されたこともあり，亜種マルウェアの開発が盛んである．IoT 機器の亜種マルウェアの増加に伴い，特に亜種に強いマルウェア検知機構が求められている．

セキュリティ対策の 1 つとしてアンチウイルスソフトの利用が挙げられる．特に亜種に強いアンチウイルスソフトとして，ヒューリスティック方式のものと，ビヘイビア方式を採用したものが挙げられる．ヒューリスティック方式は，検査対象のプログラムを静的に解析して，その特徴を抽出し，その情報からマルウェアを検知する手法である．未知のウイルスに耐性がある一方で，プログラムの静的解析を行う必要があるため，難読化を施されたマルウェアについては解析が難しいという欠点がある．ビヘイビア方式は，プログラムの挙動をリアルタイムで観察し，悪意のあるプログラムを動的に検知する手法である．この手法はマルウェアの実行ファイルでなく，その挙動を元にルールを作

[1] 豊橋技術科学大学
Toyohashi University of Technology 1-1, Hibarigaoka, Tempaku-cho, Toyoahshi-shi, Aichi, Japan
[2] 工学院大学 Kogakuin University
[3] 長崎県立大学 University of Nagasaki

成するため，亜種マルウェアに耐性がある．一方で，ルール作成の難しさに加え，プログラムを実行する必要があるため検査に時間がかかってしまうという欠点も持つ [4]．

亜種に強いヒューリスティック方式やビヘイビア方式のマルウェア検知機構では，バイナリの静的解析やプログラムの挙動の監視を行っているため，ソフトウェアでの実装が前提となっている．その一方で，IoT 機器は CPU やメモリに代表されるハードウェアリソースを多く確保することが出来ない [5]．そのため，これらのマルウェア検知機構を IoT 機器本来のアプリケーションと同時にソフトウェアとして実装することは，ハードウェアリソースの問題により難しい．そこで，その解決策としてマルウェア検知機構をハードウェアとして外部にオフロードすることを提案する．マルウェア検知機構を専用のハードウェアとして実装することで効率よくマルウェア検知処理を行う．これにより，IoT 機器のアプリケーションとセキュリティ機構のハードウェアリソースの競合を避けることを目的とする．

ハードウェアにオフロードしやすい情報としてプロセッサの情報に注目した．そこで，プロセッサ情報を特徴量とした機械学習によるマルウェアと正常プログラムの判別機構を提案する．プロセッサ情報とは，オペコードや，命令長といった情報の他，キャッシュや分岐予測機構の動作の情報などが含まれる．本稿では，ハードウェアへのオフロードに対する予備評価として，仮想マシンを用いたプロトタイプを実装し，提案手法の評価を行う．

第 2 章では関連研究について，第 3 章では提案機構のメインアイデアについて述べる．第 4 章では提案機構の実装方法について説明し，第 5 章で評価を行う．第 6 章で評価結果を示し，第 7 章で考察を行い，第 8 章で本論文をまとめる．

2. 関連研究

機械学習を用いてマルウェアを，発見する手法が既にいくつか提案されている．Arvind らは，Android に対するマルウェアが実行時に要求する権限に注目し，それを特徴量とすることでマルウェアを検知する手法を提案している [6]．また複数の機械学習手法による分類精度の比較も行っている．

村上らは，プログラムを動的解析して API コールを抽出し，その名前の n-gram を用いて，マルウェアと正常プログラムの分類を行っている [7]．n-gram とは，隣り合う N 個の単語を一塊として文章を分解する手法である．さらに，評価用のデータを学習用のデータとの類似度に基づいて選別し，選別後の評価データの検出精度を向上させる手法についても提案している．

また，マルウェアの静的解析によって抽出した特徴を用いて，機械学習によってマルウェアを分類する方法も広く行われている [8][9]．

これらの機械学習を用いた手法では，API コールや権限を監視しており，ソフトウェアとしてマシン本体に実装する必要がある．これらの手法は一般のマシンには実装可能だが，マシンリソースの問題により IoT 機器に実装することは難しいという問題がある．そこで，提案手法では，ソフトウェア的な情報ではなくハードウェアから取得可能な情報を用いてプログラムの特徴を抽出することで，ビヘイビア法をハードウェアにオフロードし，リソースの競合の問題を解決する．

セキュリティ機構をハードウェアとして実装する手法としては ARM プロセッサに実装されている TrustZone が挙げられる [10]．TrustZone は実行環境を Secure World と Normal World の 2 つに分けている．Normal World から Secure World へのアクセスは制限されるため，Secure World 上の認証情報や支払い情報をマルウェアから守ることが可能となる．また，TrustZone を利用したセキュリティ機構の研究も行われている [11][12]．この手法は，マルウェアから，重要なデータを守るための機構であり，マルウェア実行の検知を目的とした本研究とは異なる．

特定の処理を専用のハードウェアにオフロードすることで，高速な処理を実現する製品も存在する．SmartNIC はネットワークのパケットフィルタリングや暗号化などといった処理を，専用ハードウェアによって処理する技術である [13]．これによって CPU のリソースを他のアプリケーションに充てることが可能となる．また，OpenSSL の暗号化について FPGA ベースによる，専用ハードによってパフォーマンスの向上を図る研究も行われている [14][15]．

このように，特定の処理に特化したハードウェアを利用する研究が進められている．マルウェアの検知は，非常に重要な問題であるため，マルウェア検知専用のハードウェアの作成は有用である．

3. 提案手法

この章では，今回作成するプロトタイプの概要と使用したプロセッサ情報の詳細について述べる．

3.1 メインアイデア

仮想マシン上でプログラムを動作させ，その時のプロセッサ情報を取得する．その後，取得したプロセッサ情報を特徴量とした機械学習によって正常プログラムとマルウェアの判別を行う．

本稿で使用するプロセッサ情報とは，プロセッサの内部情報を指す．このプロセッサ情報には，静的なプロセッサ情報と，動的なプロセッサ情報が存在する．静的なプロセッサ情報とは，プログラムカウンタやオペコード，レジスタ番号など，バイナリから解析可能な情報を指す．動的なプロセッサ情報とは，レジスタの値など，命令実行時のプロセッサの内部状態を示す情報である．この動的なプロ

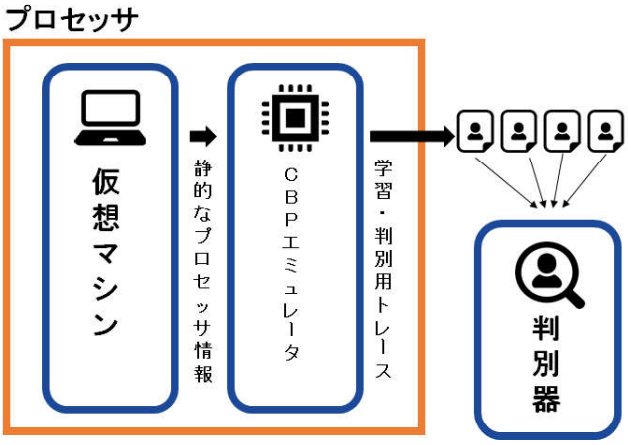

図 1　提案機構の概要
Fig. 1　Outline of proposed mechanism

表 1　特徴量として使用したプロセッサ情報
Table 1　Processor information used as a feature value

プロセッサ情報	説明
insn	命令種別
op	オペコード
cond	コンディショナルフラグ
len	命令長
dn	NOP の命令距離
dj	JUMP の命令距離
do	OTHER の命令距離
inst_cache_hit_rate	命令キャッシュの累積ヒット率
L2_cache_hit_rate	L2 キャッシュの累積ヒット率
btb_hit	BTB のヒット/ミス
direction	分岐命令の分岐方向
pred_direction	分岐命令の分岐予測方向

セッサ情報は，実際にプロセッサを動作させないと取得することが出来ない．プログラムの学習，判別にはプロセッサ情報を命令実行順に並べた情報を使用する．この情報をトレースと表記する．トレースをプログラム固有のデータとして学習や判別に利用する．

提案手法では，特に動的なプロセッサ情報である，プログラムの空間的局所性と時間的局所性に注目した．空間的局所性とは，一度参照された命令アドレスの付近が再び参照されやすいという特性であり，時間的局所性とは，一度参照された命令アドレスは短時間のうちに再び参照されやすいという特性である．空間的局所性と時間的局所性はプログラムごとに特徴的である．局所性は直近参照された命令アドレスに沿って動作する，命令キャッシュや分岐予測機構の動作に表れやすい．そこで，命令キャッシュと分岐予測機構である CBP エミュレータ (Cache/Branch Predictor エミュレータ) を作成し，これらの動作を再現することで，局所性を含むプロセッサ情報を抽出する．

図 1 に提案機構の概要図を示す．

今回作成するプロトタイプでは，本物のプロセッサの代わりに仮想マシンと，前述した CBP エミュレータを使用する．

3.2　プロセッサ情報の概要

表 1 に特徴量として使用したプロセッサ情報を示す．

表 1 における，insn から do までは，静的なプロセッサ情報である．静的なプロセッサ情報は QEMU[16] から直接取得できる情報と，そこから計算した情報に分けられる．QEMU とは，フリーの CPU エミュレータであり，本研究では仮想マシンの構築に使用する．insn から len までは，QEMU から直接取得できる情報である．insn は NOP，LOAD，JUMP，OTHER の 4 種類の命令種別を表す．dn から do は，QEMU から得られた情報から計算した情報であり命令の距離を表す．これらは，NOP，JUMP，OTHER の 3 種類の命令が最後に実行されてから何命令経過したかをあらわしており，それぞれ dn，dj，do に対応する．

inst_cache_hit_rate から pred_direction は，CBP エミュレータを介して得られる，動的なプロセッサ情報である．inst_cache_hit_rate は，命令キャッシュの累積ヒット率である．ここで，累積ヒット率とは処理された全命令に対してヒットした命令の割合をあらわす．L2_cache_hit_rate は，命令キャッシュの下位キャッシュである L2 キャッシュの累積ヒット率である．btb_hit は BTB のヒット/ミスである．ヒットの場合は 1，ミスの場合は 0，それ以外の場合は-1 となる．pred_direction は，gshare によって予測された分岐方向，direction は実際に分岐した方向である．分岐成立の場合は 1，分岐不成立の場合は 0，それ以外はどちらの値も-1 となる．

4.　実装

この章では，プロトタイプの実装について述べる．今回作成した，プロトタイプは大きく分けて以下の 3 つの要素から構成される．

- 仮想マシン
- CBP エミュレータ
- 判別器

仮想マシン部で取得できる，静的なプロセッサ情報のみで構成されるトレースを，中間トレースとする．この中間トレースに CBP エミュレータを利用して動的なプロセッサ情報を付加したトレースをトレースデータとする．学習・判別に用いるのは，このトレースデータである．

中間トレースに含まれる静的なプロセッサ情報から，CBP エミュレータで付加される動的なプロセッサ情報を計算する例を図 2 に示す．動的なプロセッサ情報として，分岐先予測機構である BTB のヒット/ミスを挙げる．BTB のヒット/ミスを計算するための静的情報として，プログラムカウンタ，オペコード，命令長がある．まず，オペコー

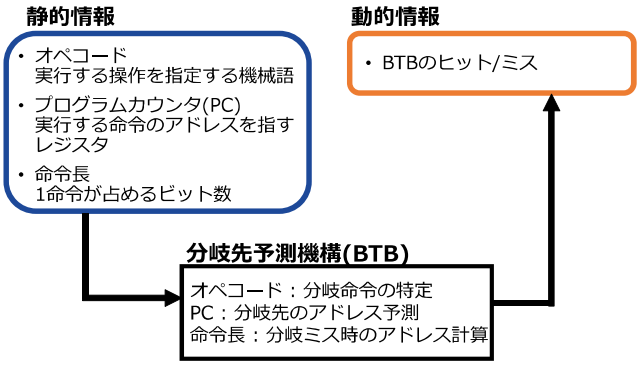

図 2　動的なプロセッサ情報の計算
Fig. 2　Calculating dynamic processor information

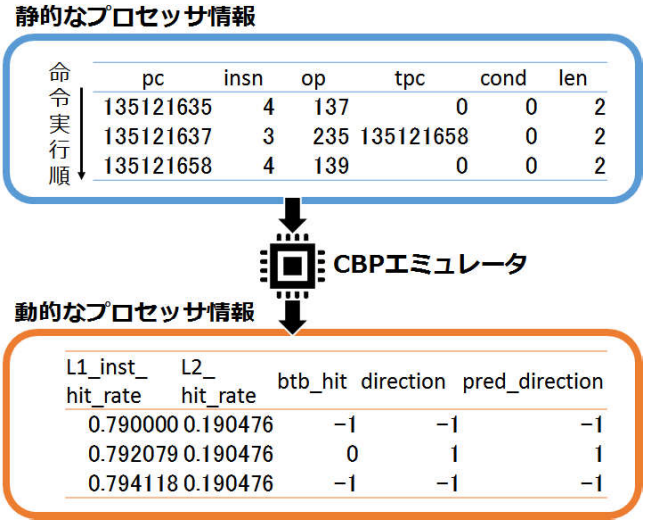

図 3　動的なプロセッサ情報の例
Fig. 3　Example of dynamic processor information

ドによって分岐命令を判別し，BTB を動作させるかを決定する．分岐命令であった場合，プログラムカウンタを元に BTB を動作させ，分岐先のアドレスを予測する．命令長は分岐ミスをした際のアドレスの計算に使用する．最終的に分岐先のアドレスが予測したものであれば BTB ヒット，そうでなければ BTB ミスの情報を得る．このような分岐予測機構の内部情報は実際に CPU を動作させないと得ることができないため，エミュレータを作成して CPU の動作を再現することで対応している．計算された動的なプロセッサ情報の例を図 3 に示す．図 3 において，プロセッサ情報は命令実行順に並んでおり，1 行が 1 命令に対応している．

4.1　仮想マシン

仮想マシンでは，プログラムを動作させ，その際の中間トレースを取得する．仮想マシンにはオープンソースのエミュレータである QEMU を使用した．QEMU の-d in_asm オプションを利用することで QEMU 上で実行されている命令のアセンブリコードを取得出来る．しかし，既存のデバッグ機能では，CBP エミュレータを動作させるために必要な，命令長の情報を取得することができない．そこで QEMU のソースコードを改変し，命令長の情報を取得できるようにすると共に，データとして扱いやすいようにログの取得部分を改変した．

また仮想マシンによる中間トレースの取得は python プログラムによって管理される．仮想マシン動作中に，外部の python プログラムによって動作開始のシグナルを送信することによって中間トレースの取得が開始される．同様に，動作終了のシグナルを送信することによって，プロセッサ情報の取得が終了する．

4.2　CBP エミュレータ

CBP エミュレータでは，仮想マシンで取得した中間トレースを使用し，命令キャッシュと分岐予測機構の動作を再現することで，仮想マシンでは取得できない動的なプロセッサ情報を付加する．CBP エミュレータは以下の 3 つの要素によって構成される

- 命令キャッシュ
- BTB
- gshare

命令キャッシュは使用頻度の高い命令を保存するためのキャッシュメモリである．BTB は過去の分岐命令の実行結果を保持し，次回以降の分岐命令の実行時にフェッチすべき命令アドレスを予測する機構である．gshare は過去の分岐命令の結果から，分岐命令が成立となるか不成立となるかを予測する機構である．これらのプロセッサの機構について，C 言語で実装を行った．

4.3　判別器

判別器では，機械学習によって，あるトレースが正常プログラムのものであるか，マルウェアのものであるか判別する．機械学習のアルゴリズムにはランダムフォレストを選択した．ランダムフォレストは，複数の決定木を組み合わせて，各決定木の予測結果の多数決によって結果を得る，アンサンブル学習の一種である．高次元のデータ分類においても効率的に動作し，高次元の特徴においても効率的な学習が可能であるという特徴を持つ．プロセッサ情報にはカテゴリ値が多いが，このような場合でもスケーリング不要である点からランダムフォレストが適していると考えた．

判別器の動作は，学習フェーズと判別フェーズに分けられる．学習フェーズでは，あらかじめ取得した正常プログラムの学習用トレースと，マルウェアの学習用トレースを使用し，判別器を作成する判別フェーズでは判別を行いたいプログラムの判別用トレースを判別器に与える．判別用トレースは 1 プロセッサ情報ごとに正常であるか異常 (マルウェア) であるか判別される．判別用トレース内にはプロセッサ情報が多く存在するが，1 命令が 1 プロセッサ情報に対応している．最終的に 1 プログラムのトレースデー

タ全体のうち，正常プログラムと判別された割合と，マルウェアと判別された割合からなる中間値が出力される．正常プログラムと判別された割合を Normal，マルウェアと判別された割合を Attack とする．Attack が一定の閾値以上であれば，プログラムはマルウェアであると判別し，そうでなければ正常プログラムであると判別する．

5. 評価

この章では評価方法及び，評価結果について述べる．

5.1 プログラムのサンプル

正常プログラムとして，CentOS6．9 にデフォルトでインストールされているコマンドのバイナリから 15 種類を選び使用した．ls， tar， cat， uname， mkdir， date， cp， ps， wc， chmod， df， grep， find のそれぞれのコマンドに対して，実行開始から実行終了までを動作の区切りとしてトレースデータを取得する．仮想マシンの電源を ON にした後，コマンド実行可能になった時点で，中間トレースの取得を開始し，被攻撃マシン上で正常プログラムを実行する．正常プログラムの動作が終了した時点で中間トレースの取得を終了する．その後，中間トレースに CBP エミュレータを用いて動的なプロセッサ情報を付加しトレースデータとする．

マルウェアとして，シェルコード作成ツール msfvenom で作成した，shell/reverse_tcp と meterpreter/ reverse_tcp の 2 種類のペイロードを実行ファイル形式にしたものを基本のマルウェアとし，それぞれ reverse_nonx_tcp と reverse_tcp_uuid に機能拡張したもの，パッキングしたもの，エンコードしたものの 10 種類を用意した．reverse_tcp は，攻撃対象のシェルを獲得するものである．被攻撃マシンを外部から操作可能にする点が IoT 機器に感染するマルウェアと似ていることからこのマルウェアを選択した．

パッキングにはフリーのパッカーである UPX[17] を利用した．なお，元ファイルのサイズが小さすぎると，UPX を利用することが出来ない．そのため，msfvenom の-n オプションを利用し，UPX の利用に十分である，4MB になるように NOP を追加し，その後パッキングを行った．エンコードは msfvenom の機能を用いて行った．今回は msfvenom 内に用意されたエンコーダの中から，QEMU を用いた実験環境で動作するものとして msfvenom から，encoder/x86/single_static_bit を選択しエンコードを行った．

正常プログラムと同様に仮想マシンの電源を ON にした後，コマンド実行可能になった時点で，中間トレースの取得を開始し，被攻撃マシン上で疑似マルウェアを実行する．攻撃マシンから被攻撃マシンのコマンドシェルが利用可能になった時点で中間トレースの取得を終了する．その後，正常プログラムと同様に，CBP エミュレータを用いて動的なプロセッサ情報を付加しトレースデータとする．

5.2 評価方法

用意したトレースデータを，学習用トレースと判別用トレースの 2 つに分ける．正常プログラムのトレースデータからは，ランダムに ls，tar，cat，uname，find の 5 種類を学習用トレースとし，それ以外を判別用トレースとした．マルウェアのトレースデータは，基本となるマルウェアである，shell/reverse_tcp と meterpreter/reverse_tcp を学習用トレースとし，それ以外を判別用トレースとした．

学習用トレースを用いて学習を行い，判別器を作成する．その後，判別を行い，中間値における Attack の値が閾値 (50%) 以上であれば，そのプログラムはマルウェア，閾値未満であれば正常プログラムと判別する．判別用データは，プログラムごとに同じ手順で，2 つずつ取得した．これは，トレースデータを取得する際に，同一プログラムで合っても，外部割込み等の影響のため，トレースの形が異なるためである．

以下に，評価の項目を示す．

- 正常プログラムの判別
- 亜種マルウェアの判別
  - 機能拡張したマルウェアの判別
    - reverse_nonx_tcp の判別
    - reverse_tcp_uuid の判別
  - 難読化したマルウェアの判別
    - パッキングされたマルウェアの判別
    - エンコードされたマルウェアの判別

6. 結果

正常プログラムの判別結果を**表 2** に，機能拡張したマルウェアの判別結果を**表 3** に，難読化したマルウェアの判別結果を**表 4** にそれぞれ示す．各プログラムはそれぞれ 2 回ずつトレースデータを取得し，判別を行った．

表 2 から，判別用トレースについて，多少の誤差はあるものの，全ての場合において，Attack の値が 50%を下回っているため，正常プログラムと判別できていることが確認できる．また，表 3 では，用意した全ての機能拡張したマルウェアについて，Attack の値が，50%の閾値を超えているためマルウェアとして判別することができている．表 4 においても，全ての難読化したマルウェアについて Attack の値が 50%を超えておりマルウェアと判別できている．学習に使用したのは，機能拡張や難読化されていない基本のマルウェアであったため，このことから亜種について耐性があることが確認できる．

既存手法との確認のために，VirusTotal[18] を用いる．VirusTotal とは，ファイルを複数のアンチウイルスエンジンで検査し，その結果を表示してくれるフリーのオンラインスキャナーである．VirusTotal には，基本マルウェア，パッキングしたマルウェア，エンコードしたマルウェアをそれぞれ与える．与えたファイルの検査結果においてマル

表 2　正常プログラムの判別結果
Table 2　Discrimination result of normal program

プログラム名	中間値 [%]		判別結果
	Normal	Attack	
chmod	95.431	4.569	Normal
	98.633	1.367	Normal
date	89.113	10.887	Normal
	88.509	11.491	Normal
df	91.061	8.939	Normal
	98.923	1.077	Normal
grep	99.571	0.429	Normal
	98.066	1.934	Normal
mkdir	98.427	1.573	Normal
	97.019	2.981	Normal
sort	99.48	0.52	Normal
	99.713	0.287	Normal
ps	91.288	8.712	Normal
	73.782	26.218	Normal
wc	98.449	1.551	Normal
	98.872	1.128	Normal
cp	99.898	0.102	Normal
	98.303	1.697	Normal
more	99.604	0.396	Normal
	99.537	0.463	Normal

表 3　機能拡張した亜種マルウェアの判別結果
Table 3　Discrimination result of extended subspecific malware

プログラム名	中間値 [%]		判別結果
	Normal	Attack	
shell/	6.777	93.223	Attack
reverse_nonx_tcp	23.640	76.360	Attack
meterpreter/	3.245	96.755	Attack
reverse_nonx_tcp	4.076	95.924	Attack
shell/	13.185	86.815	Attack
reverse_tcp_uuid	4.111	95.889	Attack
meterpreter/	2.887	97.113	Attack
reverse_tcp_uuid	3.547	96.453	Attack

表 4　難読化した亜種マルウェアの判別結果
Table 4　Discrimination result of obfuscated subspecific malware

プログラム名	中間値 [%]		判別結果
	Normal	Attack	
shell/reverse_tcp	3.524	96.476	Attack
(packing)	2.197	97.803	Attack
meterpreter/reverse_tcp	1.219	98.781	Attack
(packing)	0.969	99.031	Attack
shell/reverse_tcp	29.079	70.921	Attack
(encord)	20.893	79.107	Attack
meterpreter/reverse_tcp	5.518	94.482	Attack
(encord)	4.907	95.093	Attack

ウェアと判断したアンチウイルスエンジンの数を確認し，その数が減少していれば，各難読化処理の有効性が確認できたとする．shell と meterpreter それぞれについて，基本のマルウェア，パッキングしたもの，エンコードしたものの VirusTotal のスキャン結果を表 5 に示す．

表 5　VirusTotal のスキャン結果
Table 5　Virus Total scan results

プログラム名	難読化処理無し	パッキング	エンコード
shell	23/60	0/59	18/60
meterpreter	23/60	3/59	17/60

表 5 は，VirusTotal に登録されているアンチウイルスエンジンのうち，いくつのアンチウイルスエンジンでマルウェアを判別できるかを示している．基本マルウェアは，23 個のアンチウイルスエンジンでマルウェアを判別できているのに対し，パッキングした shell については全てのアンチウイルスエンジンで判別することができず，パッキングした meterpreter は 3 つのアンチウイルスエンジンで判別することができた．さらにエンコードしたマルウェアは shell については 18 個，meterpreter については 17 個となっている．この結果から，実験で使用した難読化ツールの UPX と msfvenom はアンチウイルスエンジンの回避にある程度有効であることが確認できる．一方で，表 4 に示したとおり，提案手法では，難読化したマルウェアが正しく判別できており，アンチウイルスエンジンを回避するようなマルウェアにも耐性があることがわかる．

7. 考察

まず，同一プログラムから作成した判別用トレースであるにも関わらず，中間値に誤差が発生する．この原因として，実行したプログラム以外の，バックグラウンドで実行されているプログラムの影響が考えられる．トレースデータを確認すると，ジャンプ命令を介さずに，突然プログラムカウンタの値が大きく変化し，一定数の命令を実行後，またもとのプログラムカウンタの値付近に戻ってくる部分が確認できた．これは OS による外部割込みの典型的な動作である．この OS の外部割込みが発生するタイミングや回数によって，キャッシュや分岐予測機構の動作が変化してしまったものと思われる．しかし，中間値は変動しているものの，閾値による判定に影響するほどではない．中間値の変動が最も大きいものが，ps の判別結果であり，Attack の値が 8.712%と 26.218%となっている．しかし閾値を 50%に設定するならばこの差は十分許容できる．

次に，CBP エミュレータのキャッシュを試行ごとに，何もデータが格納されていない初期状態で実行している点である．今回の実験では，なるべく条件を同じにするため，CBP エミュレータを起動する際は，内部に何もデータが

格納されていない状態からエミュレーションをスタートしている．一方で，実際の環境では，直前までに実行されていたプログラムのデータが命令キャッシュや分岐予測機構に格納されている．しかし，直前に実行されていたプログラムのデータは，全てキャッシュミスとなるため、なにもデータが格納されていない初期状態とほぼ同等であるとみなすことが出来る．

最後に，QEMU に存在する TB キャッシュによる影響である．QEMU は，命令を TB（Translation Block）と呼ばれる単位に区切って実行している．この TB ごとに，動的バイナリ変換を行い，一旦中間コードに変換した後，ホストのアーキテクチャの命令へと変換している．この際，QEMU は高速なエミュレーションのため，TB ごとに変換結果をキャッシュし，再利用を試みている．今回使用した，-d in_asm オプションでは，キャッシュから利用されたゲストの命令については出力されておらず，ゲスト上で実行された全ての命令について出力していない可能性が存在する．また，これに付随して QEMU の出力する情報を処理して CBP エミュレータが動作しているため，キャッシュまたは分岐予測機構を完璧にエミュレーションできているわけではない．しかし，このように一部命令がキャッシュされたとしても，同一命令の処理回数が減少するだけである．これにより，提案機構の負荷が減少する可能性がある．

8. まとめ

IoT デバイスのようにマシンリソースに制約がある機器で，亜種マルウェアを判別するために，セキュリティ機構を専用ハードウェアとしてオフロードすることを想定した．オフロードしやすい特徴としてプロセッサ情報が有効であることを確認するために，機械学習によって亜種マルウェアが判別可能であるか評価を行った．提案機構について仮想マシンを用いたプロトタイプを実装し，検証した結果，正常プログラムを正しく判別することが出来た．マルウェアについては機能拡張した亜種マルウェア，パッキングやエンコードといった難読化処理を施した亜種マルウェアの判別を行った．判別の結果，基本マルウェアの学習によって亜種マルウェアを判別することができた．今後は，より現実のハードウェアに近い状況でのシミュレーションや，実際のハードウェアへの実装を検討する．

謝辞 本研究の一部は，JSPS 科研費 17K00076，16K00071 の支援により行った．

参考文献

[1] 総務省, 平成 28 年版情報通信白書第 1 部第 2 章 IoT 時代における ICT 産業動向分析, http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h28/html/nc120000.html (Accessed 2018-01-16).
[2] P. Maki, S. Rauti, S. Hosseinzadeh, L. Koivunen, and V. Leppanen. Interface diversification in IoT operating systems, UCC '16: Proceedings of the 9th International Conference on Utility and Cloud Computing, pp.304-309, 2016.
[3] J. A. Jerkins. Motivating a market or regulatory solution to IoT insecurity with the Mirai botnet code, 2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC), pp.1-5, 2017.
[4] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh. A survey on heuristic malware detection techniques, The 5th Conference on Information and Knowledge Technology, pp.113-120, 2013.
[5] Z. K. Zhang, M. C. Y. Cho, C. W. Wang, C. W. Hsu, C. K. Chen, and S. Shieh. IoT Security: Ongoing Challenges and Research Opportunities, 2014 IEEE 7th International Conference on Service-Oriented Computing and Applications, pp.230-234, 2014.
[6] A. Mahindru and P. Singh. Dynamic Permissions based Android Malware Detection using Machine Learning Techniques, Proceedings of the 10th Innovations in Software Engineering Conference, pp.202-210, 2017.
[7] 村上純一, 鵜飼裕司. 類似度に基づいた評価データの選別によるマルウェア検知精度の向上, コンピュータセキュリティシンポジウム 2013 論文集 2013(4), pp.870-876, 2013.
[8] F. Adkins, L. Jones, M. Carlisle, and J. Upchurch. Heuristic malware detection via basic block comparison, 2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE), pp.11-18, 2013.
[9] P. Khodamoradi, M. Fazlali, F. Mardukhi, and M. Nosrati. Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms, 2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS), pp.1-6, 2015.
[10] TrustZone - Arm Developer, https://developer.arm.com/technologies/trustzone (Accessed 2018-01-29).
[11] A. M. Azab, P. Ning, J.Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp.90-102, 2014.
[12] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone, Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, pp.488-501, 2017.
[13] G. Sabin and M. Rashti. Security offload using the SmartNIC, A programmable 10 Gbps ethernet NIC, 2015 National Aerospace and Electronics Conference (NAECON), pp.273-276, 2015.
[14] A. Thiruneelakandan and T. Thirumurugan. An approach towards improved cyber security by hardware acceleration of OpenSSL cryptographic functions, 2011 International Conference on Electronics, Communication and Computing Technologies, pp.13-16, 2011.
[15] J. K. T. Chang, S. Liu, J. L. Gaudiot, and C. Liu. Hardware-assisted security mechanism: The acceleration of cryptographic operations with low hardware cost, International Performance Computing and Communications Conference, pp.327-328, 2010.
[16] QEMU. https://www.qemu.org/ (Accessed 2018-01-09).

[17] The Ultimate Packer for eXecutables. https://upx.github.io/ (Accessed 2018-01-09).
[18] VirusTotal. https://www.virustotal.com (Accessed 2018-01-09).