Computer Security Symposium 2019
21 - 24 October 2019

プロセッサ情報によるマルウェア検知における特徴量のビット数削減手法の検討

永井 雄也[1,a)]　小林 良太郎[1]　加藤 雅彦[2]　嶋田 創[3]

概要：IoT 機器の新たなセキュリティ技術の 1 つとして，我々はプロセッサ情報によるマルウェア検知機構を提案してきた．現在，我々は CPU および提案実装のハードウェア記述言語による実装を進めている．IoT 向けのハードウェアであるため，機械学習で生成する分類器を構成するために必要なハードウェア量をできる限り削減したい．そこで本稿では，特徴量の中でも重要かつ小数であるキャッシュヒット率に着目し，小数ビット列の分類に必要な部分のみを使用することで，特徴量の表現に必要となるビット数削減を試みた．ビット数の削減により，特徴量を保持するレジスタのサイズや特徴量を演算する演算器のサイズが小さくなるため，分類器のサイズが小さくなる．結果として，分類精度へ大きな影響を出さずに，少なくとも 37.5%のビット数を削減できることが確認した．また，分類精度への少々の影響を許容するトレードオフ下で，さらにビット数を削減できる余地があることも確認した．

キーワード：マルウェア検知, ハードウェア, 機械学習, IoT

Examination of bit number reduction method of feature amount in malware detection by processor information

Yuya Nagai[1,a)]　Ryotaro Kobayashi[1]　Masahiko Kato[2]　Hajime Shimada[3]

Abstract: As one of the new security technique for IoT devices, we have proposed a malware detection mechanism based on processor information. Currently, we are implementing both CPU and proposed mechanism with Hardware Description Language. Since it is hardware for IoT, we want to reduce the amount of hardware required to construct a classifier generated by machine learning as far as possible. So in this paper focuses on the cache hit rate, which is an important and represented by a decimal number of features. We tried to reduce the number of bits by using only a part of the fractional bit string. By reducing the number of bits, we can reduce the size of the classifier by reducing register size and arithmetic unit size which holds and computes features. As a result, we confirmed that the number of bits could be reduced by at least 37.5 % without affecting the classification accuracy. We also confirmed that there was room for further reduction in the number of bits by sacrificing slight classification accuracy degradation.

Keywords: Malware detection, Hardware, Machine learning, IoT

1. はじめに

2016 年に DNS プロバイダの米 dyn 社は，Mirai に感染した機器で形成されたボットネットから大規模な DDoS 攻撃を受けた [1]．また，ほぼ同時期に Mirai のソースコードが GitHub に公開された．この 2 つの出来事を堺に，IoT 機器への攻撃が活発化した．NICTER 観測レポート 2018

[1] 工学院大学 Kogakuin University 1-24-2, Nishi-shinjuku, Shinjuku-ku, Tokyo, Japan
[2] 長崎県立大学 University of Nagasaki 1-1-1 Manabino, Nishi-sonogi-gun, Nagasaki, Japan
[3] 名古屋大学 Ngoya University, Furocho, Chikusa-ku, Nagoya, Aichi, Japan
[a)] j116210@ns.kogakuin.ac.jp

© 2019 Information Processing Society of Japan

では，攻撃に使用されている IP アドレスから発信されたパケットの内，約半数が IoT 機器への攻撃パケットであったと報告されている [2].

IoT 機器が攻撃の標的になり，最終的に攻撃に利用されてしまう原因の一部は，一般的な組み込み機器よりもさらに極端な IoT 機器の特性にある．例えば，IoT 機器はハードウェアリソースを特定の機能を実現するために必要十分に抑えることで，小型化や低コスト化を実現している．しかし，ハードウェアリソースが限られているため，汎用コンピュータのようにソフトウェアとしてアンチウイルス機能を搭載することが困難である．また，設置後数年単位で動作し続けるため，ライフサイクルが長い．逆に，ライフサイクルの長さがベンダーのメンテナンス不足と相まって，脆弱性を有した状態が長期化しやすくなっている．さらに，IoT 機器は IT の知識がないユーザでも使いやすい設計となっている．しかし，ベンダーがユーザビリティを優先した結果，汎用コンピュータでは当たり前のように行われているセキュリティ対策が行われていない．

こういった IoT 機器の問題は，未だ根本的な解決に至っていない．しかし，問題を残したまま，IoT 機器は急速に普及している．普及速度はさらなる加速をみせ，2017 年から 2020 年の 3 年間で約 170 億台の増加が予測されている [3]．このままでは，攻撃の標的が増える一方である．

この問題を解決すべく，我々はプロセッサ情報によるマルウェア検知機構を提案してきた [4], [5], [6], [7]．提案機構のマルウェア検知には，機械学習を利用する．将来的に提案機構をハードウェア化し，CPU ダイに混載する．混載した CPU を IoT 機器に組み込むことで，IoT 機器へのアンチウイルス機能実装を実現する．

検知機構の核となる分類器は機械学習を用いて生成するため，利用する特徴量のサイズが大きいほど，特徴量を保持したり，計算したりするハードウェア量が増加する．ハードウェア量の増加に伴い，消費電力，生産コストも増加するため，IoT 機器向けプロセッサとして好ましくない要素が増加する．よって，分類器のサイズは重要なポイントであり，IoT 機器の特性を殺さないために，できる限り小さくすることが望ましい．ハードウェア量は概ね特徴量の保持に必要となるビット数に比例するため，ハードウェア量増大問題を緩和する方法の 1 つに，特徴量の保持に必要となるビット数を削減することが考えられる．

この問題に対して，高瀬らはサンプリングや次元削減を用いた分類器のサイズ削減手法を提案している．適切にサンプリングすることで，分類精度を維持したまま学習量を減らせること確認している．また複数ある特徴量の内, 3 つの特徴量だけで精度を維持できることを確認している [8]．しかし，残った特徴量にビット数の削減という分類器サイズ削減の余地がある．

そこで本稿では，高瀬らの手法を加味した上で，さらに分類器サイズを削減する手法として，主要な特徴量のビット数削減を提案する．提案手法適用後の分類精度への影響やビット数の削減率で評価を行い，提案手法の有効か否かの検討を行う．

第 2 章では，提案機構について述べる．第 3 章では特徴量のビット数削減手法を述べ，第 4 章で手法の評価を行う．第 5 章で本論文をまとめる．

2. 提案機構の概要と利点

IoT セキュリティの新たなる一手として，我々はプロセッサ情報によるマルウェア検知機構を提案してきた．本章では，我々の提案してきたマルウェア検知機構について，より詳細な説明を行う．

2.1 提案機構の概要

2.1.1 検知機構に使うプロセッサ情報

提案機構におけるプロセッサ情報とは，プログラム実行時にプロセッサから得られるデータを指す．我々が現在扱っているプロセッサ情報の詳細を**表 1** に示す．

扱っているプロセッサ情報は大きく分けて，CPU が持っている情報とそれらを加工して生成する情報の 2 つである．表 1 の上段は，既存の CPU が持っている情報である．オペコードやレジスタ番号などの静的なプロセッサ情報とプログラムカウンタやレジスタの値などの動的なプロセッサ情報を含んでいる．表 1 の中段と下段は，CPU の情報を利用して生成する情報である．中段の情報は，キャッシュの hit/miss の情報から算出されるキャッシュのヒット率である．下段は，オペコードを 5 種類 (NOP，LOAD，STORE，JUMP，OTHER) に大別し，各命令種別を最後に実行してから何命令経過したかを示した情報である．

2.1.2 実装の想定

2.1.2.1 提案機構の CPU ダイへの混載

CPU 内部に組み込む形で，提案機構を実装する．提案機構は上記のプロセッサ情報を利用するために，ハードウェアで実装する．ハードウェア実装には，外部から接続する，オンボード上に接続する，SoC の一部にするや CPU に混載するなど複数の実装方法が存在する．CPU の情報を直接扱うことを考慮すると，CPU の近くにおける実装方法が提案機構に適している．それゆえ，我々は提案機構を CPU に混載を前提にしている．

実際に提案機構を CPU ダイに混載した場合のイメージを**図 1** に示す．一般的な IoT 向けの SoC には，CPU コアと同時に DRAM, GPIO, UART, I2C, SPI などの汎用インターフェースの回路ブロックが混載されている．提案機構は，それらの回路ブロックと同じレベルで混載する．

2.1.2.2 RISC-V で作った独自 CPU に実装

提案機構を CPU に組み込むには，ベースとなる CPU が必要である．広く普及している x86 や ARM の CPU に

表 1 プロセッサ情報の詳細
Table 1 Details of processor information

プロセッサ情報	説明
pc	プログラムカウンタ
insn	命令種別
op	オペコード
addr	ロード/ストアアドレス又は分岐先の PC
cond	条件分岐フラグ
regnum	命令で使用するレジスタ番号
regval	regnum に格納されているレジスタの値
btb_hit	BTB による分析のヒット/ミス
direction	実際に分岐した方向
pred_direction	gshare によって予測された方向
L1_inst_hit_rate	L1 命令キャッシュの累積ヒット率
L1_data_hit_rate	L1 データキャッシュの累積ヒット率
L2_hit_rate	L2 キャッシュの累積ヒット率
dn	NOP の命令距離
dl	Load の命令距離
ds	Store の命令距離
dj	Jump の命令距離
do	その他の命令距離

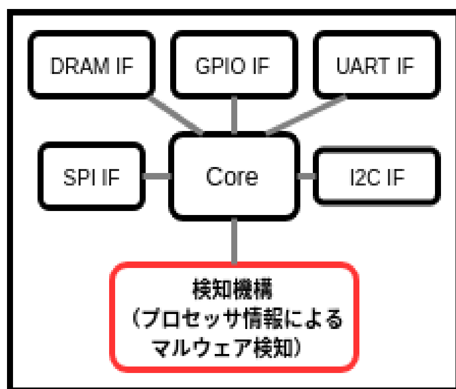

図 1 提案機構を載せた CPU ダイ
Fig. 1 CPU die with proposed mechanism

は，ライセンスの問題や命令セットアーキテクチャ (ISA: Instruction Set Architecture) の複雑さなど，様々な問題で容易に手出しができない．そこで，本研究はオープンソースの ISA である RISC-V を利用して独自実装を行った CPU に対して，提案機構を組み込む形で進めることとした．RISC-V を選択した理由は，オープンソースの ISA の中で近年開発が活発な ISA であり，コンパイラ等のソフトウェア開発環境が充実している．また，実装事例など参考にできる情報が多いためである．

2.1.3 検知機構の核となる分類器について

分類器作成には，ランダムフォレストを利用する．機械学習の特徴量として利用するプロセッサ情報は，特徴量ごとにスケールが異なる．ランダムフォレストはデータのスケールに対して不変であるため，プロセッサ情報を扱うのに適している．それゆえ，分類器の作成に利用する機械学習アルゴリズムに，ランダムフォレストを選んだ．

ランダムフォレストで 1 命令ごとのプロセッサ情報を学習することで，分類器を生成する．1 命令実行ごとのプロセッサ情報の集まりを特徴量として扱う．この特徴量用いて，ランダムフォレストで学習を行い，分類器を生成する．

2.1.4 動作の仕組み

2.1.4.1 プロセッサ情報を検知機構に送る仕組み

プロセッサ情報を検知機構に送る一部の流れを**図 2** に示す．図 2 は，命令キャッシュからプロセッサ情報を検知機構へ送るイメージである．各矢印はプロセッサ情報の流れを表す．黒は本来の流れ，青は検知機構への流れである．各ボックスは提案機構のパーツを表す．黒のボックスは CPU が元から持っているパーツ，赤のボックスは新規に追加するパーツである．

図 2 では，プロセッサ情報を検知機構に送るために，2 つのステップを踏んでいる．ステップ 1 として，命令実行時，プロセッサ情報がやり取りされるタイミングで，プロセッサ情報をバイパスして検知機構に送る．図 2 中では，命令キャッシュから個々の命令をフェッチする時に，hit/miss の情報とフェッチされた命令をバイパスし，提案機構に送る．ステップ 2 として，キャッシュのヒット率や命令距離のような加工情報を生成し，検知機構に送る．キャッシュのヒット率や命令距離は，プロセッサ情報をバイパスするだけでは扱えない．加工情報を得るには，プロセッサ情報を受け取った上で，特定の処理を加える必要がある．図 2 中では，ヒットカウンタ（キャッシュヒット率を算出するパーツ）が命令キャッシュの hit/miss 情報を受け取り，命令キャッシュヒット率を算出し，算出値を検知機構に送る．

図 2 はあくまで一部の流れである．検知に使う他のプロセッサ情報も，同様の流れで検知機構に送られる．

2.1.4.2 検知の仕組み

まず，送られてきた 1 命令ごとのプロセッサ情報に対して，分類器が判別を行いマルウェアか否かの判定を出す．各ハードウェアは 1 命令ごとにプロセッサ情報を検知機構に送る．1 命令分のプロセッサ情報が集まり次第，分類器は判別を行い，0(正常) または 1(マルウェア) の判定を出す．

次に，分類器の判定を元にして，指定区間の攻撃率を算出する．指定区間が 10,000 であれば，10,000 命令分の判定結果 (0 または 1) を加算し，10,000 で割ることで，攻撃率を算出する．

最後に，指定区間の攻撃率が設定した閾値以上であるか否かで判断を下す．攻撃判定数が閾値以上であれば，マルウェアと判定する．攻撃判定数が閾値未満であれば，正常プログラムと判定する．

2.2 IoT 機器に搭載可能

既存のマルウェア検知機構をハードウェアの制約が多い IoT 機器に搭載することは難しい．既存のマルウェア検知

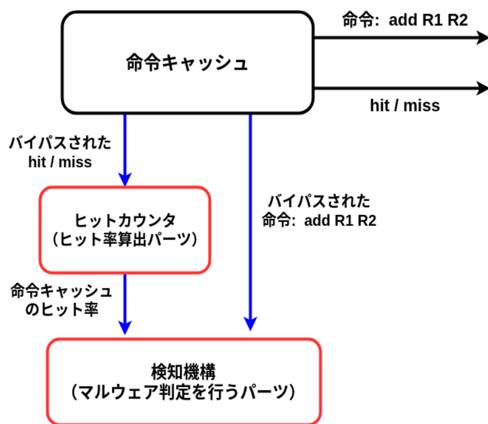

図 2 一部のプロセッサ情報の流れのイメージ図

Fig. 2 Image of part of processor information flow

機構は，ソフトウェアとしての実装を前提にしているものがほとんどである．そのため，実行時には「ソフトウェア実装されたマルウェア検知機構を実行するのに必要となるハードウェアリソース」を要求されるが，多くの IoT 機器は要求を満たすハードウェアリソースを有していない．また，CPU など IoT 機器本来の処理を行うハードウェアリソースを要求分増やすと，消費電力の増加などの問題が発生するため，リソースを増やすことも難しい．それゆえ，IoT 機器に既存のマルウェア検知機構を搭載することは難しくなっている [9].

一方で，我々の提案機構は CPU 本体とは別のハードウェアで実装するため，ハードウェアリソースの制約を気にせず搭載できる．また，提案機構をハードウェア実装に最適化することで，ソフトウェアを CPU 上で実行するよりも少ないハードウェア量や消費電力での実装が可能になる．

そのため，既存のマルウェア検知機構とは異なり，我々の提案機構であれば，ハードウェアリソースの限られた IoT 機器にも搭載できると考える．

3. 特徴量のビット数削減手法

提案機構のハードウェア化を実現するために，分類器のサイズは可能な限り小さくしたい．本章では，分類器のサイズを可能な限り小さくする方法として，特徴量のビット数削減を提案する．高瀬らの削減手法 [8] を考慮し，提案手法は各キャッシュヒット率のみに焦点を当て，常に同じ上位ビットと余分な下位ビットを削除することでビット数の削減を実現する．

3.1 ステップ 1: 常に同じ上位ビットを削除

一般的なプログラム実行において，キャッシュヒット率は値の変動が安定すると，値が一定以上に収まることが多い．キャッシュヒット率が一定の値以上の時，キャッシュヒット率を 2 進数で表現すると，小数点以降の上位ビットは常に同じ値になる．例えば，データキャッシュのヒット率が 0.75 以上のとき，2 進数上は先頭 3 ビットが必ず「11」になる．そこで，常に同じ上位ビットを削除することを考える．

小数点以降の常に同じ上位ビットを削除しても，ランダムフォレストであれば問題がない．小数点以降の常に同じビット数の削除は，スケール変換に相当する．前述のデータキャッシュのヒット率であれば，$0.75 \leq hitrate \leq 1$ が，$0 \leq hitrate \leq 1$ に変化する．スケールが変化するだけなら，我々が使っているランダムフォレストは影響を受けない．それゆえに，常に同じ上位ビットを削除しても精度に影響が出ない．

つまり，キャッシュヒット率の常に同じになっているビット列を削除することで，分類器による判別の精度を保ちつつ，ビットの削減を期待できる．

本ステップを適用した上で，次のステップ適用することで，さらにビット数を削減する．

3.2 ステップ 2: 分類に必要な上位ビット列だけ使用

分類器のサイズを小さくするためには，キャッシュヒット率を小数のまま扱うことはできない．10 進数の小数を正確に表現できないコンピュータは与えられたビットをすべて使い，近似することで小数を表現している．ビット数の観点から見ると，コンピュータ上の小数は無駄が多い．ビット数の無駄が多いなら，できるだけサイズを抑えたい分類器の特徴量として小数を使うことはできない．

そこで小数の数値そのものではなく，小数のビット列に注目する．命令キャッシュのヒット率が「0.9」だとする．「0.9」を 2 進数にすると，$0.1110\dot{0}11\dot{0}$（固定小数点表記）になる．この 10 進数小数を 2 進数で表現したビット列に注目する．

小数ビット列の内，分類に必要な小数点以降の上位ビット列のみを扱うことで，ビット数削減を期待できる．対象のビット列として $0.1001\dot{1}10\dot{1}$ があり，小数点以降の上位 8 ビットが分類器の判別に必要だとする．この場合，$0.10011101 \ldots 1101$ の内，「10011101」を使用し，残りの「1101...1101」を捨てる．もともとのビット列に 32 ビット使われていたと仮定した場合，処理によって $32 - 8 = 24$ ビット節約できたことになる．

4. 提案手法の評価

本章では，前章で提案したビット数削減手法の評価を行う．

4.1 評価環境

現存する CPU にはプロセッサ情報をバイパスする機能がないため，本評価では提案機構をエミュレーションすることで，得られたプロセッサ情報を利用した．

4.1.1 エミュレーションによるプロセッサ情報の取得方法

まず，QEMU[10] を用いてベースとなるプロセッサ情報を取得した．QEMU とは CPU エミュレータであり，プログラム実行時のエミュレート対象 CPU のプログラムカウンタ，アセンブリ及びレジスタの値を出力する機能を有する．本評価では，1 命令ごとに 3 つの情報をまとめて 1 行に出力できるように QEMU を改変した．捕獲した IoT 機器を標的としたマルウェアバイナリのほとんどが ARM バイナリだったため，QEMU の full-system emulation 機能で，Raspberry Pi と同等の仮想マシンを作った．作った仮想マシンの上でプログラムを実行することで，対象プログラム実行時のプロセッサ情報を取得した．

次に，QEMU から出力されたプロセッサ情報を特徴量として使えるように変換した．QEMU から出力された形式では，特徴量として使用することはできない．そのため，QEMU から出力されたプロセッサ情報を使用できる特徴量に変換するプログラムを Python で作成した．作成した Python プログラムに QEMU から出力されたプロセッサ情報を入力することで，pc，insn，op，addr，cond，regnum，regval，dn，dl，ds，dj，do の 12 の特徴量が生成される．

最後に，独自に作成したエミュレータで予測分岐とキャッシュヒット率を算出した．QEMU から予測分岐とキャッシュの情報は取得できなかった．そのため 2 つの情報を得るために，予測分岐とキャッシュのエミュレーションを行う，CBP（Cache / BranchPredictor）エミュレータというプログラムを作成した．このエミュレータに生成済みの pc，op，addr，cond の 4 つの特徴量を渡すことで，btb_hit，direction，pred_direction，L1_inst_hit_rate，L1_data_hit_rate，L2_hit_rate が算出される．

4.1.2 学習データと評価データ

エミュレーションによって得られたプログラムごとのプロセッサ情報の集まりを，本節では，トレースデータと表現する．

4.1.2.1 使用したトレースデータ

攻撃のトレースデータとして使用したマルウェアの詳細を**表 2** に示す．2018 年 7 月〜12 月の間でハニーポット Cowrie[11] を使って，捕獲したマルウェアの内，動作が確認でき，100,000 以上命令数を取得できたマルウェアを使用した．マルウェアの詳細情報は，VirusTotal[12] でファイルのハッシュ値を使って検索をかけ，取得した．マルウェアのラベル付けは，一貫性を持たせるために Symantec の判定結果のみを参考にした．命令数はエミュレーション環境でマルウェアを実行し，得られた命令の数である．

正常のトレースデータとして使用したプログラムの詳細を**表 3** に示す．本評価では Web サーバのミドルウェアとして，多く利用されている Apache 及び Tomcat を正常プログラムとして使用した．命令数は，エミュレーション環境上で，HTML ファイル対する外部からのアクセスを処理した際に，得られた命令の数である．トレースデータごとにアクセスする HTML を変えることで，トレースデータに違いをつけた．

4.1.2.2 学習データ

学習データとして使用したトレースデータを**表 4** に示す．2 つの分類器を作るために，学習データは 2 パターン用意した．トレースデータの重複がないように，それぞれ選択した．

値の変動が安定した以降の各キャッシュのヒット率を使用するために，対象のトレースデータの先頭 10,000 命令を削除した．作成したキャッシュヒット率を算出するプログラムでは，累積数が 0 で，キャッシュが空の状態からエミュレーションが始まる．値が安定するには，ある程度累積数が貯まるのを待つ必要がある．どのトレースデータでも，多くとも 10,000 命令前後で値が安定していた．それゆえ，学習に使うトレースデータの先頭 10,000 命令を削除した．

対象のトレースデータにサンプリングと次元削除を行った．高瀬らのサンプリング手法 [8] を利用して，プログラムごとに 4,000 命令（全体を 400 個に分割して，先頭の 10 命令を抽出）になるようにサンプリングを行った．また，寄与率の高いことが判明している各キャッシュのヒット率 [8] だけを使用するように次元削除を行った．

4.1.2.3 評価データ

評価に使用した攻撃と正常のトレースデータを**表 5** に示す．攻撃と正常ともに，学習に使用したトレースデータと重複がないように選んだ．評価データに使用したトレースデータにも，先頭 10,000 命令の削除と次元削除を行った．

4.2 評価方法

4.2.1 ステップごとの分類精度を取得

各ステップ適用後の分類精度への影響を知るために，適用前，ステップ 1 適用後，ステップ 1, 2 適用後ごとに分類器を作成し，評価データに対する分類を行い，分類精度を算出した．

4.2.1.1 ステップ 1 を学習と評価データに適用

CBP エミュレータでは，キャッシュヒット率の値が安定すると，命令キャッシュのヒット率が 0.9 以上，データキャッシュのヒット率が 0.8 以上，L2 キャッシュのヒット率が 0.5 以上だった．そのため，本評価では命令キャッシュのヒット率は先頭 3 ビット，データキャッシュのヒット率は先頭 2 ビット，L2 キャッシュのヒット率は先頭 1 ビット削除した．

常に同じ上位ビットの削除した数値は，適用後の数値を y，変換対象のヒット率を a，削除した上位ビット数を b とすると，

表 2　使用したマルウェアの一覧
Table 2　List of malware used

Name	VirusTotal Result	Symantec Analysis Result	File Type	Packer	命令数
Mirai1	27/56	Linux.Mirai	ELF	none	108,421
Mirai2	27/28	Linux.Mirai	ELF	none	101,248
Mirai3	25/54	Linux.Mirai	ELF	none	177,347
Mirai4	27/57	Linux.Mirai	ELF	none	102,202
Trojan1	32/58	Downloader.Trojan	Shell Script	none	788,146
Trojan2	23/60	Downloader.Trojan	Shell Script	none	779,080
Trojan3	17/54	Trojan.Gen.NPE	Shell Script	none	869,080
Trojan4	29/57	Trojan.Gen.NPE	Shell Script	none	840,647

表 3　正常プログラム一覧
Table 3　List of normal programs

プログラム	命令数
apache1	1,378,077
apache2	118,742
apache3	1,562,983
apache4	1,330,948
tomcat1	2,764,277
tomcat2	2,429,322
tomcat3	2,835,481
tomcat4	2,410,959

表 4　分類器一覧
Table 4　List of classifiers

Name	Learning Trace
分類器 1	Mirai1
	Trojan1
	apache1
	tomcat1
分類器 2	Mirai2
	Trojan2
	apache2
	tomcat2

表 5　評価データ一覧
Table 5　Evaluation data list

Type	Trace Name
攻撃	Mirai3
	Mirai4
	Trojan3
	Trojan4
正常	apache3
	apache4
	tomcat4
	tomcat4

$$x = a \times 2^b$$
$$y = x - \lfloor x \rfloor$$

で算出した．

4.2.1.2　ステップ 2 を学習と評価データに適用

32 ビット使用したときの値を，ステップ 2 適用後の基準の分類精度として使用した．ハードウェア化したときのキャッシュヒット率の扱いは，まだ決まっていない．本評価では，ハードウェア化したとき，キャッシュヒット率を 32 ビットの固定小数点で表現すると仮定した．それゆえ，32 ビット時の分類精度を基準にした．

必要な上位ビット列だけを使用した数値は，適用後の数値を y，変換対象のヒット率を a，使用ビット数を b とすると，

$$y = \lfloor a \times 2^b \rfloor$$

で算出した．

4.2.1.3　分類器を生成

提案手法適用前，ステップ 1 適用後，ステップ 1, 2 適用後の学習データを使い，それぞれの分類器 1，2 を生成した．分類器の生成には，sckit-learn の sklearn.ensemble.RandomForestClassifier を使用した．パラメータには，再現性を確保するためにシード = 1 のみを設定した．

4.2.1.4　分類精度の算出

生成した 3 パターンの分類器 1, 2 で同じ処理を行った評価データに対する分類精度を算出した．分類のクラスを a，分類結果を x，分類精度を y とすると，

$$a = \begin{cases} 0 & (normal) \\ 1 & (attack) \end{cases}$$
$$x = (a_1, a_2, \cdots, a_n)$$
$$y = \frac{1}{N}\sum_{i=0}^{N} x_i$$

で分類精度を算出した．攻撃の評価データに対して，正しく分類できれば，値が大きくなる．正常の評価データに対して，正しく分類できれば，値が小さくなる．

4.2.2　使用ビットごとの分類精度を取得

必要最低限のビット数を知るために，ステップ 1 を適用して上で，使用上位ビットを 32〜1 の範囲で変え，それぞれのビット数での分類精度を算出した．ステップ 1, 2 の適用，分類器の生成および分類精度の算出は，前節と同様の手順で行った．

4.3　評価結果

4.3.1　ステップごとの分類精度への影響

適用前の分類精度を基準とした，各ステップ適用後の分類器による評価データに対する分類精度への影響を**表 6** に示す．適用前とステップ 1 適用後を比較して，影響の最大値を赤，適用前とステップ 1，2 適用後を比較して，影響の最大値を緑で色付けしている．

4.3.2　使用ビットごとの分類精度

どちらの分類器でも少なくとも 20 ビットあれば，32 ビッ

表 6　手法適用後の分類精度への影響

Table 6　Impact on classification accuracy after method adaptation

評価データ	分類器 1			分類器 2		
	適用前 (基準)	削除後	削除，整数化後	適用前 (基準)	削除後	削除，整数化後
Mirai3	0.948	+0.004	0	0.952	0	0
Mirai4	0.976	0	+0.023	0.761	+0.018	+0.004
Trojan3	0.906	+0.010	+0.028	0.891	0	0
Trojan4	0.980	-0.005	0	0.986	0	0
apache3	0.038	-0.001	+0.002	0.060	+0.001	-0.030
apache4	0.016	0	+0.012	0.022	+0.005	-0.021
tomcat3	0.033	0	0	0.028	+0.002	-0.015
tomcat4	0.107	-0.001	-0.005	0.181	+0.001	-0.001

表 7　変化の分岐点となるビット数

Table 7　Number of bits that are the branching point of change

評価データ	分類器 1	分類器 2
	ビット数	ビット数
Mirai3	15	18
Mirai4	15	18
Trojan3	15	8
Trojan4	14	8
apache3	19	20
apache4	16	20
tomcat3	10	18
tomcat4	15	18

トと同じ分類精度を保てていた．32 ビット時の分類精度を基準とし，基準の値との差が 0.001 以上となるビット数との境目になっていたビット数を**表 7** に示す．

どちらの分類器でもビット数を 2 ビット以下まで減らしてしまうと，半数以上の評価データに対して，0.000 や 1.000 になった．分類器 1 では，2 ビットだとすべての攻撃評価データに 1.000 であり，1 ビットだと Mirai3 に対して 1.000 で，apache3，4 と tomcat3 に対して 0.000 であった．分類器 2 では，2 ビットだとすべての攻撃評価データに 1.000 であり，1 ビットだと Mirai3 と Trojan3，4 に対して 1.000 で，Mirai4 と tomcat3 に対して 0.000 であった．

使用ビットを少なくするほど，分類精度への影響範囲が大きくなり，一部では基準値から 0.100 以上の差になった．2 ビット以下を除いた，20 ビット以降の分類精度への影響の最大値とその時のビット数を**表 8** に示す．最大値に同値があった場合のビット数は，値の大きい方を表示している．

4.4　考察

4.4.1　提案機構の有効性

提案手法の適用した場合の分類精度への影響は軽微であった．ステップ 1 適用後の分類精度への影響は，分類器 1, 2 で 0.020 未満であり，ステップ 1，2 適用後は，分類器 1, 2 で 0.030 未満であった．提案機構では分類器による判定結果は中間値であるため，最終判断を下す閾値の調整次第で，0.030 未満の影響は許容できる．つまり，提案機構の分類器サイズを削減するために，提案手法を適用しても問題がないと言える．

提案手法を適用すれば，少なくともビット数を 37.5 %削減できる．分類器 1, 2 で 20 ビットまでは，すべての評価データに対して，32 ビット使用時と同じ分類精度を保持できた．つまり分類器に関係なく，多くとも 20 ビット使えば，元々のキャッシュヒット率と同じ分類精度が得られると考えられる．多くとも 20 ビット使用すれば良いならば，提案手法を適用することで，最低でも 37.5%ビット数を削減できる．

使用ビット数を分類器のパラメータとして，3〜19 ビットの範囲で調整を行えば，さらにビット数を削減できる．分類器ごとに調整が必要だが，20 ビットよりビット数を削れる．20 ビット以降でも，多くの場合で，32 ビット使用時の分類精度との差が 0.100 未満であった．一方で，32 ビット使用時の分類精度との差が 0.100 以上になる可能性があることもわかった．提案機構は 0.100 未満の差を許容できても，0.100 以上の差は，許容しきれない可能性がある．そのため，使用ビットを 20 ビット未満にするときは，分類器ごとに慎重な調整が必要である．ただし，使用ビットを 2 ビット以下にするべきはでない．2 ビット以下では 0.000 や 1.000 が多く見れられた．0.000 や 1.000 がでると，機械学習では過剰適合もしくは適合不足が疑われる．それゆえ，ビット数を減らすためでも，2 ビット以下にするべきではない．

4.4.2　分類器サイズについて

分類器サイズを削減するために，特徴量のビット数を削減した．主目的が分類器サイズ削減であるならば，提案手法の有効性の議論の中で，ビット数を削減した際の分類器サイズについて触れるべきである．しかし，現状，分類器をハードウェア化することができていないため，ビット数を減らした場合の分類器サイズを計測することができな

表 8　20 ビット以降の分類精度への最大の影響
Table 8 Maximum impact on classification accuracy after 20 bits

評価データ	分類器 1			分類器 2		
	32 ビット時 (基準)	最大の影響	ビット数	32 ビット時 (基準)	最大の影響	ビット数
Mirai3	0.948	+0.034	9	0.952	-0.050	17
Mirai4	0.999	-0.026	8	0.765	+0.169	4
Trojan3	0.934	-0.039	14	0.891	+0.070	4
Trojan4	0.980	-0.036	3	0.986	-0.017	4
apache3	0.040	-0.015	3	0.030	+0.038	5
apache4	0.028	-0.063	3	0.001	+0.094	3
tomcat3	0.033	-0.030	7	0.013	+0.031	5
tomcat4	0.102	+0.042	6	0.180	+0.025	5

かった．そのため，有効性の議論はビット数の削減率に対する言及でとどまっている．

4.4.3　キャッシュヒット率の算出区間の調整について

ハードウェア化した際の演算を考慮して，提案手法は左右のシフト演算のみで処理できるようになっている．しかし，本評価で扱った各キャッシュヒット率は，ハードウェア化した際の演算が全く考慮されていない．各キャッシュヒット率も，算出区間を 2 のべき乗するなど調整を行えば，ハードウェアにとってやさしい演算で算出できる可能性がある．つまり，ハードウェア化を考慮するなら，各キャッシュヒット率の算出区間を調整するべきであると考える．

5.　まとめ

本稿では，提案機構のハードウェア化を実現するために，分類器サイズをできる限り小さくする方法として，特徴量のビット数削減手法を提案し，検討・評価を行った．結果として，提案手法は有効であり，元の特徴量と比べて，少なくとも 37.5 %のビット数を削減できることを確認した．使用する上位ビット数を分類器のパラメータとして，調整を行うことで，更にビット数を削る余地があることも確認した．以上の評価により，さらなる分類器のサイズ削減が期待できる．

評価を通して，新たな課題も見つかった．見つかった課題は，

- ビット数削減後の正確な分類器サイズの計測
- キャッシュヒット率の算出区間の調整

の 2 つである．

今後も提案機構の実現を目指して，新たに見つけた課題に取り組んでゆく．

謝辞　本研究の一部は，JSPS 科研費 17K00076，19K11968，19H04108 の支援により行った．

参考文献

[1] Dyn: Dyn Analysis Summary Of Friday October 21 Attack, Dyn (online), available from ⟨https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/⟩ (accessed 2019-08-13).

[2] サイバーセキュリティ研究室：NICTER 観測レポート 2018，国立研究開発法人　情報通信研究機構　サイバーセキュリティ研究所（オンライン），入手先 ⟨https://www.nict.go.jp/cyber/report/NICTER_report_2018.pdf⟩（参照 2019-08-13）．

[3] 総務省：平成 30 年版情報通信白書，総務省（オンライン），入手先 ⟨http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h30/html/nd111200.html⟩（参照 2019-07-28）．

[4] 小林良太郎，高瀬　誉，大谷元輝，大村　廉，加藤雅彦：機械学習を用いたプロセッサレベルでのプログラム分類に関する予備評価，研究報告コンピュータセキュリティ，Vol. 117, No. 316, pp. 5–10 (2017).

[5] 大谷元輝，高瀬　誉，小林良太郎，加藤雅彦：プロセッサレベルの特徴量に着目した亜種マルウェアの検知，研究報告コンピュータセキュリティ，Vol. 2018-CSEC-80, No. 31, pp. 1–8 (2018).

[6] 小池一樹，小林良太郎，加藤雅彦：Windows におけるプロセッサレベルの特徴量に着目した亜種マルウェアの検知，コンピュータセキュリティシンポジウム 2018 論文集，pp. 593–600 (2018).

[7] 鈴木庸介，小林良太郎，加藤雅彦：Windows におけるプロセッサレベルの特徴量に着目した亜種マルウェアの検知，コンピュータセキュリティシンポジウム 2018 論文集，pp. 875–881 (2018).

[8] 高瀬　誉, 小林良太郎, 加藤雅彦, 大村　廉：プロセッサ情報を用いたマルウェア検知機構における分類器のサイズ削減手法の検討, 情報処理学会研究報告, Vol. 2018-CSEC-83, No. 9, pp. 1–8 (2018).

[9] Zhang, Z. K., Cho, M. C. Y., Wang, C. W., Hsu, C. W., Chen, C. and Shieh, S.: IoT Security: Ongoing Challenges and Research Opportunities, *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, pp. 230–234 (2014).

[10] QEMU: (online), available from ⟨https://www.qemu.org/⟩ (accessed 2019-08-14).

[11] Cowrie: Cowrie SSH/Telnet Honeypot, (online), available from ⟨https://github.com/cowrie/cowrie⟩ (accessed 2019-08-15).

[12] VirusTotal: (online), available from ⟨https://www.virustotal.com/⟩ (accessed 2019-08-15).