Computer Security Symposium 2021
26 - 29 October 2021

IoT マルウェアにおける関数の依存関係と結合の順序関係に基づくライブラリ関数名の特定

赤羽 秀[1,*]　川古谷 裕平[2]　岩村 誠[2]　岡本 剛[1]

概要： IoT 機器を標的とした攻撃の増加に伴い，IoT 機器で動作するマルウェアが増加している．多くの IoT マルウェアはライブラリ関数を静的に結合し，シンボル情報を消去しているため，関数名による解析が困難である．我々の先行研究では，パターンマッチングにより，10 種類のアーキテクチャの IoT マルウェアの構築に使用されたすべてのツールチェインと静的結合されたライブラリ関数の 91.7%を特定した．残り 8.3%の関数は関数の候補を特定したが，これらの候補から関数を特定するには静的解析が必要であった．特に解析者にとって馴染みのないアーキテクチャの検体を静的解析する負担は大きい．そこで，本研究ではライブラリ関数の依存関係と結合の順序関係を手がかりにして関数の候補から関数を特定する手法を提案する．実験では提案手法による関数の特定精度の評価を行い，マルウェアに静的結合された 99.8%のライブラリ関数の名前を特定した．さらに，他の手法と比較し，提案手法の特定精度が他の手法より高いことを確認した．最後に特定した関数名のリストを使ってマルウェアの分布を可視化し，関数名リストでマルウェアを分類できる可能性を示した．

キーワード：関数名特定，ライブラリ関数特定，パターンマッチング，Linux マルウェア解析，マルウェア分類

Identifying Library Function Names Based on Function Dependencies and Linking Ordering in IoT Malware

Shu Akabane[1,*]　Yuhei Kawakoya[2]　Makoto Iwamura[2]　Takeshi Okamoto[1]

Abstract: Much IoT malware includes static linking of library functions, and their symbols such as function names are stripped hindering function-level analysis. We previously showed that pattern matching could identify 91.7% of library functions statically linked to IoT malware. For the remaining 8.3% of library functions, we identified their candidates, but static malware analysis was required to identify functions from these candidates. In particular, static malware analysis of an architecture with which the analyst is unfamiliar is a heavy burden. In this paper, we propose a method to identify a function from candidate functions based on the dependency relationship of functions and the order of their linking. In experiments, our method identified 99.8% of the names of all library functions in 3,983 samples. Furthermore, we compared the accuracy of our method with other methods and confirmed that the accuracy of our method is higher than other methods.

Keywords: Function name identification, library function identification, pattern matching, Linux malware analysis, malware classification

1. 序論

ネットワークに接続された組み込み機器を標的とした攻撃が増加しており，それに伴い組み込み機器で動作するマルウェアが増加している[1]．性能が低い機器でも安定して動く安定性や，環境構築の容易さにより，多くの安価なネットワーク機器や IoT 機器が Linux を使用している．従来は，デスクトップ機器を標的とする攻撃が一般的であったため，多くのデスクトップ機器で使用されていた Windows で動作するマルウェアの解析技術が求められていた．このため，マルウェアの解析技術は Windows を中心に高度に洗練されていった．しかし，現在増加している IoT 機器で使用されている Linux は，Windows と比べて対応している CPU アーキテクチャの種類が多い上に，実行ファイルの形式も異なるため，IoT マルウェアの解析技術は未だ十分に洗練されていない．

マルウェア解析では，マルウェアが呼び出す関数名等のシンボル情報が特定できる場合，その関数を既知のライブラリ関数として扱えるので，その関数の静的解析など更なる解析の手間を省略できる．しかし，ライブラリ関数がプログラムに静的結合されている場合，関数名等のシンボル情報は消去されていることが多い[2, 3]．このため，マルウェアによって呼び出される関数を特定するためには，更なる解析が必要であり，それには各アーキテクチャのマシンコードを読み解く知識や時間が必要である．したがって，IoT マルウェアに静的結合されたライブラリ関数の特定手法が求められている．

今日までに，関数の特定手法として様々な手法が提案さ

1 神奈川工科大学
Kanagawa Institute of Technology
2 NTT 社会情報研究所
NTT Social Informatics Laboratories

* s1822071@cco.kanagawa-it.ac.jp

れている．関数の特定手法は，パターンマッチングによる手法（例えば，[4]など）と制御フローグラフの類似度による手法（例えば，[5-7]など）に分類される．パターンマッチングによる手法では，関数を正確に特定できる反面，解析対象の検体が使用するコンパイラやライブラリの組み合わせ（ツールチェイン）の変化の影響を受けやすい．制御フローの類似度による手法では，ツールチェインの構成要素が変化した場合であっても，使用するライブラリのプログラムコードが大きく変化しない限り，制御フローは変化しないため，ツールチェインの構成要素を細かく特定しなくても関数を特定できる場合がある．しかし，システムコール番号などのデータが 1 バイトしか違いがない関数を誤検知することがある．実際に，ライブラリ関数の中にはこのような関数がいくつか存在する．

関数を特定するため，我々はパターンマッチングによって静的結合されたライブラリ関数のアドレスとその関数名を特定する手法について研究を行ってきた．はじめに，独自に収集した Intel 80386 アーキテクチャの IoT マルウェア検体に対して，パターンマッチングを 2 段階にわけるなどの工夫によって 92.5%のライブラリ関数名を特定し，残りの 7.5%の関数も関数名の候補を 4 つ程度まで絞り込むことができた[8]．つぎに，10 種のアーキテクチャの IoT マルウェア検体に対してパターンマッチングを行い，すべての検体の構築に使用されたツールチェインとすべての検体に静的結合されたライブラリ関数のアドレスを特定した[3]．また，同時にアドレスを特定した関数のうち 91.7%の関数名を特定した．これによって，ほとんどのマルウェア定義関数の機能解析等の解析が可能になったが，残り 8.3%の関数名が特定できていない関数を呼び出すケースでは，静的解析を行い，関数名が候補のうちのどれかを特定する必要があった．さらに，マルウェアが使用するライブラリ関数名のリストを利用することによってマルウェア分類[9]やマルウェア検知[10]の可能性が示されているが，これらには完全な関数名のリストが必要であった．

本研究では，複数のライブラリ関数の候補の中から正しい関数名を特定するため，ライブラリ関数の依存関係と結合時の順序関係に基づいた関数名の特定手法を提案する．多くのライブラリ関数は，関数内で別のライブラリ関数を呼び出す．このため，ライブラリ関数の結合時には，結合するライブラリ関数が呼び出す関数も実行ファイルに結合される．このようにライブラリ関数には依存関係がある．依存関係を持つライブラリ関数が特定できたら，この依存関係により関数名を絞り込める可能性がある．さらに，ライブラリ関数はオブジェクトファイルのエントリ毎に結合されるため，結合するライブラリ関数と同じエントリ内で定義された内部関数も結合される．このため，ライブラリ関数の結合順には，エントリ内の順序関係がある．この順序関係により関数名を絞り込める可能性がある．本研究で提案する関数名の特定手法が IoT マルウェアの関数名の特定に有効であるかを明らかにするために，ハニーポットで収集した 3,983 検体（10 種のアーキテクチャの検体を含む）に対して関数名の特定精度の評価を行った．本研究の成果は以下の通りである．

- パターンマッチングのみによる特定では，関数名を 91.7%しか特定できなかったのに対して，提案手法を適用することにより，99.8%の関数の関数名を特定した．
- 提案手法では特定できない関数とその原因と課題を明らかにした．
- 提案手法と他の手法（IDA F.L.I.R..T と BinDiff）についての関数名の特定精度を評価した結果，提案手法は 99.5%，IDA F.L.I.R.T.は 86.9%，BinDiff は 88.2%であり，提案手法の精度が最も高かった．
- 各検体の関数名リストを使って検体の分布を可視化することにより、関数名のリストが検体の分類に使える可能性を示した．
- 関数名の特定に使用した IoT マルウェア検体のライブラリ関数名の特定結果を GitHub に公開した[14]．

2. ライブラリ関数名の絞り込み

パターンマッチングによる関数名の特定では，1 つのライブラリ関数のパターンが複数のライブラリ関数と一致することがある．これが原因でライブラリ関数に複数の候補が残り，これらの関数を特定するには静的解析が必要であった．

1 つの関数のパターンが複数のライブラリ関数と一致する原因は，リンカーによるものと関数の長さによるものがある．前者はリンカーによる関数アドレスの再配置（変更）である．後者は jmp 命令や ret 命令のみなど関数が極めて短いときである．サンク関数やライブラリ関数のローカル関数（内部関数）に多く見られる．

複数の候補の中から関数名を特定するには，各ライブラリ関数の特徴を手がかりに，関数名を特定する必要がある．多くのライブラリ関数は，関数内で別のライブラリ関数を呼び出す．このため，ライブラリ関数の結合時には，そのライブラリ関数が呼び出す関数も結合される．このようにライブラリ関数には依存関係がある．依存関係を持つライブラリ関数が特定できたら，この依存関係により関数名を絞り込める可能性がある．さらに，ライブラリ関数はオブジェクトファイルのエントリ毎に結合されるため，結合するライブラリ関数と同じエントリ内で定義された内部関数も結合される．このため，ライブラリ関数の結合順には，エントリ内の順序関係がある．この順序関係により関数名を絞り込める可能性がある．内部関数は外部のライブラリ関数と依存関係がないため，この順序関係が重要な手がかりになる．本研究では，ライブラリ関数の依存関係と結合

時の順序関係を手がかりにして関数名を特定する．依存関係と結合の順序関係を手がかりにした関数名の特定はすでに特定できた関数名を手がかりにするため，1 回の処理ではすべての関数を特定できない．1 回の処理で 1 つも関数名を特定できなくなるまで特定を繰り返し行う．

2.1 依存関係を手がかりにした関数名の特定

依存関係を手がかりにした関数名の特定手法を述べる前に本手法の理解に必要な静的結合の仕組み[11]について述べる．

ライブラリ関数は関数内で別のライブラリ関数を呼び出すことが多い．このため，結合時にはライブラリ関数が使用する別のライブラリ関数も結合される．このような依存関係を解決するために，リンカーはライブラリ関数の結合時に，静的ライブラリ内の結合するライブラリ関数のシンボルテーブルと再配置テーブルを確認し，依存関係の解決を行う．リンカーは，はじめに，シンボルテーブルを参照し，依存関係にあるライブラリ関数を結合する．つぎに，再配置テーブルを参照し，結合した別のライブラリ関数を正しく呼び出すために，再配置アドレスを書き換える．このようにして依存関係を解決してライブラリ関数を静的結合する．

ここから，関数名の特定手法を述べる．依存関係を手がかりにした関数名の特定手法では，はじめに検体の構築に使用されたツールチェインが使用する静的ライブラリ（.a ファイル）を解析することで，依存関係を構築する．具体的には，静的ライブラリ内の各ライブラリ関数のオブジェクトファイルに記録されている情報（静的ライブラリ内の各関数が呼び出す関数名と，その関数の再配置アドレス）から関数の依存関係を構築する．つぎに，検体に静的結合されたライブラリ関数の依存関係を解析する．最後に，静的ライブラリから構築した関数の依存関係と，検体に結合されたライブラリ関数から構築した関数の依存関係を照合し，関数名を特定する．関数名の特定手順を以下に示す．

(1) 静的ライブラリ関数の依存関係の構築
(2) 検体のライブラリ関数の依存関係の構築
(3) 依存関係の照合による関数名の特定

2.1.1 静的ライブラリ関数の依存関係の構築

事前準備として，静的ライブラリからライブラリ関数の依存関係を構築する．依存関係による関数名の特定では，依存関係にある関数の名前と，その関数の再配置アドレスを照合し，関数名を特定する．静的ライブラリの依存関係の構築では，はじめに，静的ライブラリ内の各ライブラリ関数が定義されているオブジェクトファイルを順番に取り出す．つぎに，取り出したオブジェクトファイルのシンボルテーブルと再配置テーブルを参照し，ライブラリ関数の依存関係の構築を行う．各ライブラリ関数が呼び出す別のライブラリ関数の関数名は，シンボルテーブルを参照することで取得する．また，各ライブラリ関数が別のライブラリ関数を呼び出すために結合時に書き換えられる再配置アドレスは，再配置テーブルを参照することで取得できる．

しかし，シンボルテーブルや再配置テーブルには，変数名などの関数名以外の情報も記録されている．検体から構築した関数の依存関係には関数以外の情報は含まれないため，静的ライブラリから構築した依存関係に関数以外の情報が含まれている場合，たとえ同一の関数であったとしても，静的ライブラリから構築した依存関係と検体から構築した依存関係に不整合が発生して，関数名を特定できないことがある．提案手法では，ライブラリ関数の依存関係の構築に条件を設け，条件に一致した情報のみを使用して依存関係の構築を行った．それらの条件を以下に示す．

- 同一オブジェクト内で定義されている別の関数を呼び出すために使用される，関数属性（STT_FUNC）を持ち，セクションのインデックス番号が割り振られている情報
- 別のオブジェクト内で定義されている別の関数を呼び出すために使用される，未定義属性（SHN_UNDEF）を持つ情報
- 別のオブジェクトからも参照可能であるが，参照されない限り隠される，外部参照属性（STB_GLOBAL）属性と隠し属性（STV_HIDDEN）を合わせ持つ情報

2.1.2 検体のライブラリ関数の依存関係の構築

静的結合されたライブラリ関数の依存関係の構築では，前節で述べたとおり，依存関係にある関数の名前と，その関数の再配置アドレスを特定する必要がある．しかし，検体にはシンボル情報が残されていないことが多いため，検体に静的結合されたライブラリ関数の依存関係の構築では，関数名はパターンマッチングによる関数名の特定結果から取得した．関数の再配置アドレスは，検体を逆アセンブルして，表 1 に示した各アーキテクチャの命令のオペランドから取得した．

2.1.3 依存関係の照合による関数名の特定

依存関係の照合による関数名の特定では，静的ライブラリと検体から構築した依存関係を照合して関数名を特定する．関数名の特定では，未特定関数の呼び出し元関数の依存関係を用いた特定（Caller base）と，未特定関数が呼び出す関数の依存関係を用いた特定（Callee base）の 2 つの手法で特定する．

Caller base では，未特定関数の呼び出し元の関数の名前が特定できている場合にのみ，呼び出し元の関数から構築した依存関係を手がかりに，関数名の特定を行う．はじめに，静的ライブラリから構築した呼び出し元の関数の依存関係の中から，呼び出し元関数が呼び出す関数の名前と，

表 1 再配置アドレスとして取得する命令

アーキテクチャ	条件
ARC	bl, b.d, bl,d 命令のオペランド
ARM32	b, bl, bx, blls, blne 命令のオペランド
MIPS, MIPS64	lw, ld 命令のオペランド
Motorola m68k	bsr.s, bsr.w, bsr.l, bsr.b 命令のオペランド
PowerPC	b, bl 命令
Renesas sh4	mov.l 命令
SPARC	call 命令のオペランド
Intel 80386, x86-64	call, jmp 命令のオペランド

その関数の再配置アドレスを取得する．次に，検体に結合された呼び出し元の関数の依存関係の中から，呼び出し元関数が呼び出す未特定関数の関数名の候補と，その関数の再配置アドレスを取得する．最後に，呼び出し元関数から呼び出される関数名と一致する関数名が未特定関数の候補に存在するか，また，再配置アドレスが一致するかを確認する．すべてが一致した場合にのみ，関数名を特定したとする．

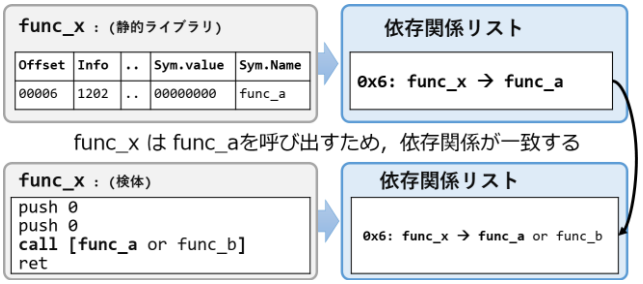

図 1 Caller base の関数名の特定

Callee base では，未特定関数が呼び出すすべての関数の名前が特定できている場合にのみ，未特定関数の候補すべての関数の依存関係と，未特定関数が呼び出す関数の名前を手がかりに，関数名の特定を行う．はじめに，未特定関数の候補すべての関数の依存関係を静的ライブラリから取得する．次に，検体に結合された未特定関数の依存関係を取得する．最後に，未特定関数の候補すべての関数から取得した依存関係と未特定関数から取得した依存関係を比較し，依存関係がすべて一致する関数が存在するかを確認する．依存関係がすべて一致した場合，関数名を特定したとする．

2.2 結合の順序関係を手がかりにした関数名の特定

ライブラリ関数を結合する際には，結合するライブラリ関数だけでなく，そのライブラリ関数が使用する内部関数や，別のライブラリ関数も結合される．このとき結合される関数は，ツールチェインに組み込まれた静的ライブラリの種類やバージョンによって変化する．よって，ライブラリ関数の順序関係には，ツールチェインごとに一定の規則

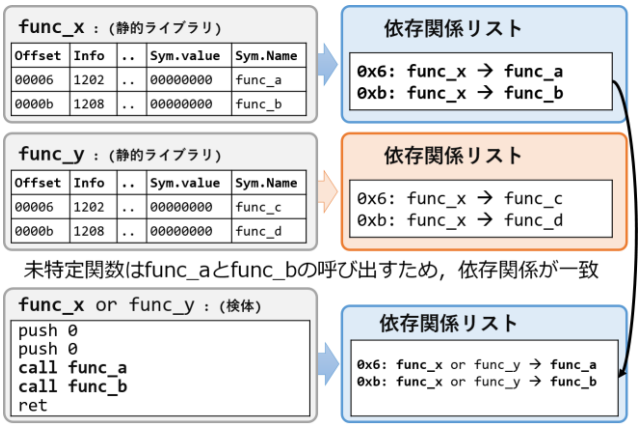

図 2 Callee base の関数名の特定

性がある．パターンマッチングによって特定されたすべての関数を含むダミープログラム（ライブラリ関数の参照のみで構成した C 言語のソースコード）を自動生成して，検体のビルドに使用されたツールチェインでダミープログラムをビルドすることで，内部関数などを含む結合の順序関係を取得できる．関数名を特定できていない関数の前後の関数の名前が特定できた場合，結合の順序関係と照合することで，関数名の候補から，順序関係が一致する関数名を特定できる．

ライブラリ関数の結合の順序関係を手がかりにした関数名の特定では，ダミープログラムの結合順と検体で特定した関数の結合順を照合し，関数名を特定する．関数名の特定手順を以下に示す．

(1) ダミープログラムの生成と関数の結合順の取得
(2) 検体に結合されたライブラリ関数の結合順の取得
(3) 結合順の順序関係の照合による関数名の特定

2.2.1 ダミープログラムの生成と関数の結合順の取得

ダミープログラムの生成と関数の結合順の取得では，ダミープログラムを生成し，ダミープログラムから関数の結合順を取得する．はじめに，パターンマッチングで特定したライブラリ関数名をすべて取得する．つぎに，ライブラリ関数の内部関数を除いてユーザー定義関数から呼び出し可能なすべてのライブラリ関数の参照（例えば，void* ptr = accept）を含むソースコードを作成し，検体のビルドに使用されたツールチェインを使って，ダミープログラムを生成する．最後に，ダミープログラムに結合された関数のアドレスと関数名を抽出し，アドレス順に並び替えを行い，結合順を取得する．アドレスと関数名の抽出には nm コマンドを使用した．

2.2.2 検体に結合されたライブラリ関数の結合順の取得

検体に結合されたすべての関数の結合順の取得では，パターンマッチングや依存関係によって特定した関数をアドレス順に並び替えた結合順を取得する．この結合順には関

数名の候補が複数ある関数も含まれる．

2.2.3 結合順の順序関係の照合による関数名の特定

結合順の依存関係の照合による関数名の特定では，ダミープログラムから取得した結合順と，検体から取得した結合順を照合し，関数名を特定する．はじめに，未特定関数の関数候補とその前後に結合された関数名を取得する．つぎに，未特定関数の前後に結合された関数名を，ダミープログラムの結合順の中から取得する．最後に，取得した関数の間に未特定関数の候補が含まれていた場合，関数名を特定できたと判断する．

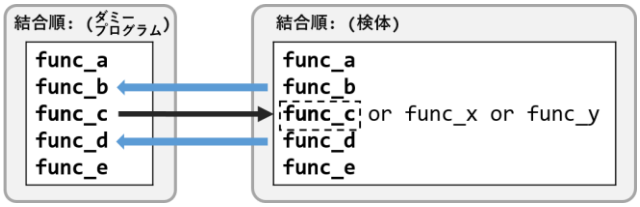

図 3 結合順の順序関係の照合による関数名の特定

2.3 関数名を特定しない関数

本研究では，パターンマッチングにより関数名の候補が複数となった関数に対して，ライブラリ関数の依存関係と結合の順序関係を手がかりにして関数名を特定する．関数名の候補が複数存在する関数であっても，依存関係と順序関係が手がかりにならない関数は，関数名の特定を行わない．関数名の特定を行わない関数を以下に示す．

- 他のライブラリ関数と依存関係がなく，結合後も他のライブラリ関数と結合の順序関係がない関数
- 参照する静的変数(データ)のみが異なり，結合後に他のライブラリ関数と結合の順序関係がない関数
  例）tolower, toupper

3. 関数候補の特定結果

3.1 性能評価に使用する IoT マルウェア検体

提案手法の有効性を明らかにするために， 2017 年 8 月から 2020 年 10 月までの期間に収集した 3,983 検体のライブラリ関数を特定することにした．これらの検体は文献[3]で使用したものと同じであるため，全検体の各アーキテクチャの割合など詳細な内訳は文献[3]に記載している．

3.2 関数名の特定結果

提案手法の有効性を評価するため，パターンマッチングのみによる特定とパターンマッチング後に提案手法を組み合わせた特定の結果を比較した．特定の結果とは，すべての検体に含まれるすべての関数について特定した個数と，すべての関数の中から検体間で重複した関数を除いたユニークな関数を特定した個数である．その結果を表 2 に示す．数値の上段はすべての関数の結果，下段はユニークな関数の結果である．

性能評価に使用した 3,983 検体には 1 検体あたり平均で 115.8 個の関数が含まれ，すべての検体には合計 461,138 個の関数が含まれる．重複を除いたユニークなライブラリ関数の個数は 1,178 個である．文献[3]のパターンマッチングにより特定できた関数の個数は 422,933 個であり，全体の 91.7%を占める．ユニークな関数の個数は 938 個であり，全体の 79.6%を占める．パターンマッチングと提案手法を組み合わせた手法により特定できた関数の個数は 460,208 個であり，全体の 99.8%を占める．パターンマッチングのみと比べて，追加で 37,275 個の関数の関数名を特定できた．ユニークな関数の個数は 1,163 個であり，全体の 98.7%を占める．パターンマッチングと比べて追加で 225 個の関数を特定できた．1 検体当たりでライブラリ関数を特定できた割合は 99.8%である．

表 2 関数名の特定結果

手法	特定した個数	未特定の個数
パターン	422,933(91.7%)	38,205(8.3%)
マッチング	938(79.6%)	240(20.4%)
パターン	460,208(99.8%)	930(0.2%)
マッチング	1,163(98.7%)	15(1.3%)
+提案手法		※仕様上の未特定個数
		673(0.1%)
		1(0.08%)

3.2.1 特定できなかった関数の原因と影響

表 2 の通り，解析対象の検体に静的結合された全ライブラリ関数のうち，0.2%の関数は候補から関数名を特定できなかった．提案手法では特定できなかった関数候補とその原因を表 3 に示す．

関数名がアンダースコアから始まる関数は，ライブラリ関数と同一のオブジェクトで定義された内部関数であり，ライブラリ関数の結合時に結合される．このため，実際には呼び出されない関数も存在する．多くの場合，例外処理やスレッド間での同期処理，呼び出し元のライブラリ関数固有の処理のいずれかを行う．これらの関数の呼び出し元のライブラリ関数名が特定できている場合，呼び出し元関数を手がかりにすればユーザー定義関数の機能分析は十分に行えるため，解析に与える影響は少ないと考えられる．関数名がアンダースコア以外から始まる関数の候補名は，いずれも stpcpy 関数と strcpy 関数のように関数名が似ている．これらの関数が行う処理は類似しており，同じ目的で使用されていることが多い．このため，これらの関数も解析に与える影響は少ないと考えられる．また，これらの関数の多くはすべての検体において候補の一方しか使われていなかったことから，使用頻度の高い関数を選択してもよいかもしれない．

表 3 提案手法では特定できなかった関数とその原因

原因	特定できなかった関数名と関数候補
a	• _Unwind_DebugHook, __cilkrts_cilkscreen_establish_c_stack, __cilkrts_destroy_worker_sysdep, __cilkrts_establish_c_stack, __cilkrts_hyperobject_noop_destroy, __cilkrts_init_worker_sysdep, __cilkrts_metacall, __cilkrts_mutex_destroy, __clear_cache, __enable_execute_stack, atomic_signal_fence, dummy_function, ignore_handler_s
	• __aeabi_errno_addr, __errno_location, __h_errno_location
	• __errno_location, __h_errno_location, __libc_pthread_init
	• __earrno_location, __h_errno_location, __libc_pthread_init, __pthread_once_fork_parent, __pthread_once_fork_prepare
	• chdir, chroot
	• execl, execlp
	• fgets, fgetws
	• fputs, fputws
	• fputs, fputws, putwc
	• sigaddset, sigdelset
b	• base_from_cb_data, base_from_object
	• feof, ferror
	• stpcpy, strcpy
c	• tolower, toupper
d	• _L_unlock_382, _L_unlock_90

a 関数の呼び出し情報の確認が困難であり，関数間の依存関係の検証ができないため
b 他のライブラリ関数と依存関係がなく，結合順序が同じであるため
c 他のライブラリ関数と依存関係がなく，参照する静的変数(データ)のみが異なるため
d 呼び出し元の関数を結合できず，結合順の検証ができないため

3.2.2 提案手法の仕様上の理由で特定しなかった関数

提案手法の仕様上の理由で関数名の特定を行わなかった関数が 1 つだけ含まれていた（673 検体で確認されている）．その関数は tolower 関数また toupper 関数である．この 2 つの関数は，ライブラリ関数からは呼び出されることがなく，ユーザー定義関数からのみ呼び出され，いずれを結合しても順序関係も同じであるため，関数名を特定しなかった．

この 2 つの関数は，参照する静的変数（データ）のみが異なる．この関数を特定するには，各ライブラリ関数が使用する静的変数に対してパターンマッチングを行う必要がある．

4. 関連研究の比較

マルウェア解析などでライブラリ関数を特定するため様々な方法が研究されてきた．本節では提案手法に関連する研究を紹介して，その中からパターンマッチングの代表的な手法として IDA F.L.I.R.T. [4]を，制御フローグラフの類似度による手法の代表的な手法として BinDiff [5]を選び，提案手法とこれらの関数名の特定精度について比較評価する．

IDA F.L.I.R.T.は，IDA Pro を提供する Hex-rays が開発した高速な関数特定技術であり，IDA Pro のライセンスで利用できる．関数の先頭 32 バイトのバイト列と関数全体のバイト列のチェックサムをシグネチャとしてマッチングを行う．ただし，チェックサムは再配置されるアドレスが含まれる場合，最初の再配置アドレス以降のバイト列をチェックサムに加えない．提案手法と同様に，IDA F.L.I.R.T.はツールチェイン毎に事前にシグネチャを生成する必要がある．

libc-database [12] は様々な Linux ディストリビューションの libc に含まれる関数のデータベースを用いて libc の Linux ディストリビューションとライブラリの特定を行える．libc のシンボル情報を活用することで，いくつかの関数の名前とアドレスがわかればライブラリを特定できる．しかし，IoT マルウェアは関数の名前やアドレスなどのシンボル情報が削除されていることが多いため，libc-database でライブラリを特定することは難しい．

FCatalog [13] は，関数の命令列の類似度を比較することによって関数を特定する．命令の順番が変化した場合や代替命令の置換の場合であっても関数を特定できる可能性がある．しかし，命令が使用するレジスタが違う場合でも同

一の命令とみなすため，関数を誤検知することが多い．文献[6]によれば，次に紹介する BinDiff や BinSequence などと比べて関数の特定精度が低いことがわかっているので，提案手法との比較は行わない．

BinDiff は，IDA Pro や Ghidra に組み込める無償のツールであり，セキュリティパッチの解析などを目的として，コールグラフの特徴量を比較して関数の差分を抽出する．ツールチェインの組み合わせにより命令が変化した場合でもコールフローの特徴量に変化がなければ類似性の高い関数として関数を特定できる．無償で提供されているため，関数特定のベンチマークとしてよく比較評価されている．

さらに，BinSequence [6] や BinShape [7]は命令列の類似度と制御フローグラフの類似度を組み合わせて関数の特定を行う．BinSequence は，BinDiff など 4 つの手法の中で最も高い精度で関数を特定できることを示している．しかし，BinDiff などと同様に BinSequence は関数の命令列や制御フローグラフの類似度が高い場合に，異なる関数を同じ関数として誤検知する可能性がある．具体的には，2 章の冒頭で述べた「1 つのライブラリ関数のパターンが複数のライブラリ関数と一致する関数」は命令列と制御フローグラフが完全に一致するため，これらの手法は提案手法が特定した関数を特定することは難しいと考えられる．また，BinSequence と BinShape の実装が公開されていないため，これらの手法と提案手法を定量的に比較することが難しい．

4.1 IDA F.L.I.R.T.との比較

提案手法と IDA F.L.I.R.T.の比較では，精度を評価するには正しい関数名が明らかである必要があるため，シンボル情報が含まれる 150 検体を用いた．ただし，MIPS など一部の検体では IDA F.L.I.R.T.がシグネチャ生成時にエラーが発生するため，これらの検体は 150 検体には含まれない．提案手法と IDA F.L.I.R.T.の特定精度を比較した結果，関数名の特定精度は，提案手法は 99.5%，IDA F.L.I.R.T.は 86.9%であり，提案手法は 12.6 ポイント高い結果となった．これは，IDA F.L.I.R.T.が誤検知を防ぐために，命令が短い関数を対象としないことや，CRC の対象コードが関数全体ではなく，関数の先頭から最初の再配置命令までしか対象としないことが原因である．

4.2 BinDiff との比較

BinDiff は 2 つの実行ファイルに対して関数の命令や制御フローの差分を確認するツールであり，提案手法のように静的ライブラリと検体の関数を比較できない．BinDiff で特定精度を評価するため，静的ライブラリの代わりに 2.2.1 項で述べた方法で，検体が使用するライブラリ関数を結合したダミープログラム（検証用プログラム）を使用することにした．BinDiff の評価が不利にならないように検体のビルドに使用されたツールチェインで検証用プログラムをビルドすることにした．つまり，検体が使用するライブラリ関数は検証用プログラムにすべて含まれている．

比較に用いる検体はシンボル情報が含まれる検体の中から検体のビルドに使用されたツールチェイン毎に 1 検体ずつ選択して，合計 21 検体を使用し，BinDiff の特定精度を算出した． BinDiff の特定精度は 88.2%であり，提案手法と BinDiff を比較して提案手法は 11.3 ポイント高い結果となった．これは，ライブラリ関数の中には execve 関数と readlink 関数のように命令列がすべて同じ上に，制御フローまでも同じ関数が多数存在するためである．

5. 関数名リストによるマルウェアの分類

Windows マルウェアなどではマルウェアが使用する API（関数）の情報によりマルウェアファミリの分類が可能であることが示されている[9]．IoT マルウェアにおいても関数名のリストによりマルウェア検知の可能性が示されている[10]．関数名リストによるマルウェア分類はアーキテクチャに依存しない利点があり，多数のアーキテクチャに存在する IoT マルウェアの分類には強みがあると考えられる．そこで，本研究では IoT マルウェアを関数名のリストだけでもマルウェアファミリを分類できるかを調べる．

関数名リストによるマルウェア分類には完全な関数リストが必要になる．しかし，3 章の結果では 15 個の関数を 1 つの関数に絞り込めなかったので，15 個の関数はすべての検体の中で最も使用頻度の高かった関数を関数の候補から選択することにした．なお，8 つの未特定関数の関数候補はすべての検体においていずれか 1 つの関数しか使用されていなかったため，これらの未特定関数は特定できた可能性が高いと考えられる．

提案手法では内部関数や依存関係のあるすべてのライブラリ関数を特定したが，内部関数や依存関係のある関数はツールチェインに依存していることがある．そのため，マルウェアのソースコードが同一であっても，ツールチェインが変化すれば関数名の組み合わせが異なることがある．このようなツールチェインの影響をなくすため，ユーザー定義関数が呼び出すライブラリ関数（マルウェアのソースコードに記載されたと予想される関数）に限定した．その結果，すべての検体において関数名リストは 563 種類に集約でき，すべての検体の関数名リストに含まれる全関数は 182 個であった．563 種類の関数名リストの出現頻度を確認した結果，出現頻度の上位 6 件の関数名リストが検体全体の 50%を占めていた．

次に関数名のリストでマルウェアファミリを分類できるかどうかを調べるため，関数名リストを各関数の有無で 0 または 1 の値からなる 182 次元のベクトルに変換して，t-SNE（t-Distributed Stochastic Neighbor Embedding）で検体の分布を可視化した．その結果を図 4 に示す．図の凡例は VirusTotal の結果を AVClass で特定したマルウェアファミリ名である．凡例の「SINGLETON」は特定のファミリに分類できなかった検体である．各ファミリに複数のクラスタ

が存在するが，各クラスタに異なるファミリが含まれる個数は少ないことから，関数名のリストでマルウェアファミリを分類できると考えられる. Mirai ファミリはいくつかのクラスタに分かれたので，検体を Mirai に限定して，アンチウイルスエンジンの ESET-NOD32 が分類した Mirai の亜種ラベルで色分けした結果を図 5 に示す. 亜種ラベルは 156 種類ある．5 つのクラスタでは特定の亜種が多数含まれたことから，関数名リストからベンダーの分類結果と関連性があることを確認した.

6. 結論

本研究では，ライブラリ関数の依存関係と結合の順序関係を手がかりにして関数名を特定する手法を提案した．パターンマッチングのみでは関数名が特定できなかったライブラリ関数に対して，提案手法による関数名の特定を行った結果，97.6%の関数名を特定した．提案手法では特定できない関数とその原因を明らかにし，これらの関数が解析時に与える影響は小さいことを示した．関数の特定や推定の手法として代表的な IDA F.L.I.R.T.と BinDiff による関数名の特定精度を比較評価した結果，提案手法は 99.5%，IDA F.L.I.R.T.は 86.9%，BinDiff は 88.2%であり，提案手法の特定精度が最も高い結果となった．最後に提案手法で特定した関数名のリストにより IoT マルウェアを分類できる可能性を示した.

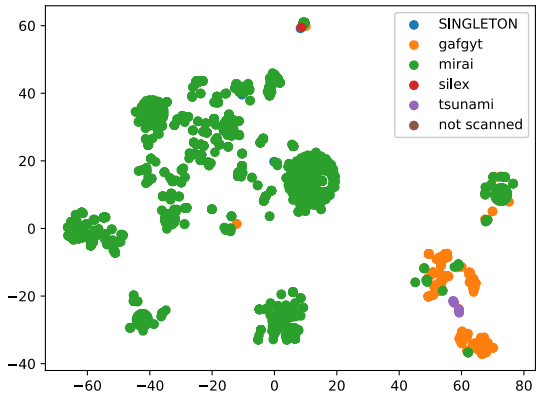

図 4 関数名リストによるマルウェアファミリの分布

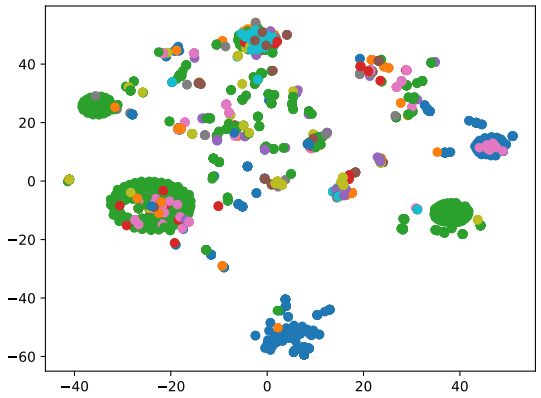
図 5 関数名のリストによる Mirai の亜種

今後の課題として，横浜国立大学の吉岡研究室が提供する IoT マルウェアの大規模データセット[15]に対する提案手法の有効性評価が挙げられる．我々のパターンマッチング手法により，51,948 検体（Intel 80386，ARM，MIPS）のうち，99.98%の検体のツールチェインを特定できる可能性があることを確認している.

参考文献

[1] SonicWall. “New SonicWall Research Finds Aggressive Growth in Ransomware, Rise in IoT Attacks”. https://www.sonicwall.com/news/new-sonicwall-research-finds-aggressive-growth-in-ransomware-rise-in-iot-attacks/, (参照 2021-03-20).

[2] Emanuele, C. and Mariano, G. Yanick, F. Davide, B.. Understanding Linux Malware. IEEE Symposium on Security and Privacy, 2018, p. 161-175.

[3] 赤羽 秀. 岡本 剛. ライブラリ関数が静的結合された IoT マルウェアのビルドに使用されたツールチェインの特定 . 研究報告コンピュータセキュリティ(CSEC), 2021, p. 1-8.

[4] Hex-Rays. “IDA F.L.I.R.T. technology: in-depth”. https://www.hex-rays.com/products/ida/tech/flirt/in_depth,/ (参照 2020-04-20).

[5] Thomas, D. and Rolf, R.. Graph-Based Comparison of Executable Objects. Symposium sur la sécurité des technologies de l’information et des communications, 2005, p. 1-8.

[6] Huang, H. and Yousser, A, M. Mourad, D.. BinSequence: Fast, Accurate and Scalable Binary CodeReuse Detection. ACM on Asia Conference on Computer and Communications, 2017, vol. 17, p. 155-166.

[7] Shirani, P. and Wang, L. Debbabi, M.. BinShape: Scalable and Robust Binary Library Function Identification Using Function Shape. International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, 2017, p. 301-324.

[8] 赤羽 秀. 岡本 剛. シンボル情報が消去された IoT マルウェアに静的結合されたライブラリ関数の特定. コンピュータセキュリティシンポジウム論文集, 2020, p. 543-550.

[9] 朝長 秀誠 . “Import API と Fuzzy Hashing でマルウェアを分類する~impfuzzy~ (2016-05-09)”. https://blogs.jpcert.or.jp/ja/2016/05/impfuzzy.html/, (参照 2020-07-15).

[10] イボット アリジャン. 大山 恵弘. 動的シンボル情報を用いた Linux 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論文集, 2019, p. 1329-1335.

[11] 坂井 弘亮. リンカ・ローダ実践開発テクニック. CQ 出版社. 2010, p17.

[12] Karlsruhe Institute for Technology CTF Team. “libc-database”. https://kitctf.de/tools/, (参照 2020-04-20).

[13] xorpd. “FCatalog”. https://www.xorpd.net/pages/fcatalog.html/, (参照 2020-04-20).

[14] Akabane, S. Okamoto, T.. “stelftools”, https://github.com/shuakabane/stelftools/, (参照 2020-06-01).

[15] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, and Tsutomu Matsumoto, Takahiro Kasama, Christian Rossow, “IoTPOT: Analysing the Rise of IoT Compromises,” 9th USENIX Workshop on Offensive Technologies (USENIX WOOT 2015), 2015.