Computer Security Symposium 2020
26 - 29 October 2020

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

赤羽 秀[1,*]　岡本 剛[1]

概要：IoT機器を標的としたLinuxマルウェアの増加とともに，ライブラリ関数を静的結合したLinuxマルウェアが多数確認されている．このようなマルウェアの多くにおいて関数などのシンボル情報が消去されているため，関数レベルでのマルウェア解析が困難であることがわかっている．そこで，関数レベルの解析を支援するため，パターンマッチングによりシンボル情報が消去されたIoTマルウェア（Intel 80386のELFファイル）に静的結合されたライブラリ関数の特定を試みた．その結果，収集した2,148検体において各マルウェアが使用する全ライブラリ関数の名前とアドレスを特定できた．ライブラリ関数の特定により，検体のビルドに使用されたツールチェインも特定でき，それらはたったの5種類であり，4種類のツールチェインはWebサイト上で公開されているものであった．VirusTotalの分類結果によれば，分析した検体には2つのファミリ（Mirai と Gafgyt）しか存在しなかったが，特定したライブラリ関数の名前リストのSHA2に基づいて検体を分類した結果、およそ263種類の亜種が存在することがわかった．

キーワード：ライブラリ関数特定，ツールチェイン特定，Linux マルウェア解析，マルウェア分類，パターンマッチング

Identification of library functions statically linked to stripped IoT malware

Shu Akabane[1,*]　Takeshi Okamoto[1]

Abstract: Many Linux malware have been found to have statically linked library functions. Much of this malware are stripped of function names and addresses, hindering function-level analysis. For function-level analysis, we identified library functions statically linked to stripped IoT malware with the Intel 80386 architecture by matching patterns. The pattern matching identified all library functions for the 2,148 samples we collected. Only five toolchains had been used to build the samples, and the five toolchains are available on the Internet. The C library used by the malware was uClibc in 98.6% of the samples and musl in 1.4%. VirusTotal classified the samples into only two malware families, i.e., Mirai and Gafgyt, but we found approximately 263 different variants by classifying the samples based on the SHA2 hash of the library function name list.

Keywords: Library function identification, toolchain identification, Linux malware analysis, malware classification, pattern matching

1. 序論

2016年頃からマルウェアに感染したIoT機器によるサイバー攻撃が世界中で確認されており，Kaspersky社の報告では2019年の上半期のみで一億件の攻撃を検知したと報告している[1]．また，多くのIoT機器が組み込み機器用のLinuxを利用しているため，Linuxマルウェアの急速な増加によって，Linuxマルウェアの解析の需要が増加している．

マルウェア解析は表層解析，動的解析，静的解析に分けられる．表層解析では，ファイルに含まれる文字列やシンボル情報などから過去に発見されたマルウェアとの類似性を調べ，解析の手掛かりを得る．また，インポートされた関数の名前リストを活用し，マルウェアファミリを分類することも可能である[2][3]．動的解析では，表層解析で得られた手掛かりをもとにサンドボックスなどでマルウェアの振る舞いを解析する．静的解析では，逆アセンブラを使って動的解析で観測されなかった機能や耐解析処理が原因で観測できなかった機能を解析する．

Linuxマルウェアにはライブラリ関数を静的結合した上にシンボル情報を削除したマルウェアが多数見つかっている[4]．このようなマルウェアには関数の名前とアドレスが含まれないため，関数レベルの解析が困難である．例えば，表層解析ではマルウェアが使用する関数名を取得できないため，関数名を手がかりにした解析ができない．動的解析では静的結合されたプログラムに対して関数レベルのトレースができない．静的解析ではライブラリ関数の特定にはマシンコードを読み解く必要があるため，解析に多くの時間を要する．様々なLinuxマルウェアに関する最新の包括的な分析でさえ，静的結合されたライブラリ関数の解析は行われていない[4]．

1 神奈川工科大学
Kanagawa Institute of Technology
* shu.akabane141@gmail.com

関数の特定にはこれまで様々な方法が提案されている．パターンマッチングによって関数を特定する方法は，コンパイラやライブラリの組み合わせ（ツールチェイン）ごとにマシンコードが変化し，組み合わせが多様であるため，関数を特定することが難しい．マシンコードや制御フローグラフの類似度によって関数を特定する方法は，オペランドのレジスタや即値が異なる場合でもその周辺のマシンコードが一致する場合，関数を誤検知する．特にライブラリ関数にはオペランドの即値のみが異なる関数が多く存在するため，誤検知が発生しやすい．

多くの IoT マルウェアは組み込み機器向けのクロスコンパイラや C ライブラリなどのツールチェインを使っていることがわかっている [4]. ツールチェインとは，システムやソフトウェアの開発に使用するツール群の総称である．さらに，Linux マルウェアは解析対策をあまり行っていないため [4]，解析妨害の目的でカスタマイズされたツールチェインではなく，よく知られたツールチェインを使っている可能性が高いと考えられる．我々の事前の調査によって Mirai のインストールガイド [5] で紹介されたツールチェインが多くのマルウェアのビルドに利用されていたことを確認している．

Ubuntu などサーバ・デスクトップ向け Linux ディストリビューションの C ライブラリは多様性が高いため，パターンマッチングによりライブラリ関数を特定するには大量の C ライブラリからパターンを生成する必要があり，それをするには多大な時間と労力が必要になる．一方，組み込み向け C ライブラリは種類が少なく，サーバ・デスクトップ向け Linux ディストリビューションの C ライブラリと比べてバージョン履歴が短い．さらに，Mirai のインストールガイド [5] で紹介されたツールチェインがマルウェア開発者に好まれているため，数種類の C ライブラリからパターンを生成するだけで，パターンマッチングによりライブラリ関数を特定できる可能性がある．

多数の Mirai が見つかっているため [1][4]， 本研究では Mirai のインストールガイドに記載されていたツールチェインに含まれる C ライブラリから生成したパターンを用いて関数を特定する．また，ウェブサイトの検索上位にランキングされている人気のツールチェインも加える．上述の方法でライブラリ関数を特定できることを確認するため，ハニーポットで独自に収集した 2,148 個のマルウェア検体でライブラリ関数とツールチェインの特定を試みる．これらの検体はすべてライブラリ関数が静的結合され，シンボル情報が削除されている．ライブラリ関数とツールチェインを特定できたマルウェアの分析を行ない，マルウェアが使用する C ライブラリと開発言語の割合や，ツールチェインとマルウェアファミリの関係性，マルウェアの収集日と使用されたツールチェインのリリース日の関係性を明らかにする. また, 特定したライブラリ関数の情報を利用して，マルウェアの分類を試みる．

本研究で得られた結果は以下の通りである．

- ツールチェイン・ライブラリ関数を特定するために必要な YARA ルールを生成した（Github に公開した [6]）．
- パターンマッチングによりすべての検体のライブラリ関数とツールチェインを特定した．
- すべての検体はたった 5 種類のツールチェインでビルドされ，4 種類のツールチェインはカスタマイズされたものではなく, Web サイト上で公開されているものであった．
- 97.4%の検体は Mirai のインストールガイドに記載されていた Firmware Linux 0.9.6 で構築したツールチェインでビルドしていた．
- すべての検体が C 言語でビルドされ, C ライブラリは 98.6%が uClibc，1.4%が musl であった．
- VirusTotal の分類結果によれば，分析した検体には 2 つのファミリ（Mirai と Gafgyt）しか存在しなかったが, 特定したライブラリ関数の名前リストのハッシュ値に基づいて検体を分類した結果，SHA2 では約 263 種類の亜種が存在し，TLSH では約 105 種類の亜種が存在した．
- マルウェアのダウンロード日とマルウェアに使用されたツールチェインのリリース日との間には相関がなかった．

2. 関連研究

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．IDA F.L.I.R.T [7] は，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．関数の先頭付近に再配置アドレスがある関数は他の関数と識別することが難しいため，IDA F.L.I.R.T はこれらの関数を特定しない. これが原因で特定できない関数が多い．また，事前にシグネチャを生成する必要があり，シグネチャは Windows のライブラリや主要な Linux ディストリビューションしか公開されていない．

libc-database [8] 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，マルウェア解析で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．また libc-database は組み込み機器向けの Linux の libc を対象として

いない.

KISS [9] は libc-database のような大規模なデータベースと IDA F.L.I.R.T.の高速な検索機能を組み合わせることにより効率的なパターンマッチングで関数を特定する. しかし, 静的結合されたプログラムでは, 17.2%の関数しか特定できなかった.

FCatalog [10] は, 関数の機械語命令列の類似度を比較し関数を特定する. 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.

また, BinDiff [11]は, 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.

さらに, BinSequence [12] や BinShape [13]は命令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した. BinShape は主要ないくつかオープンソースライブラリやアプリケーションの関数を平均で 89%の精度で特定した. しかし, BinDiff などと同様に BinSequence は関数の命令列や制御フロー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. また, BinSequence と BinShape の実装が公開されていないため, これらの手法による結果と本研究の結果の比較評価が難しい.

3. データセット

マルウェアに静的結合されたライブラリ関数を特定できるかを明らかにするために, マルウェアを収集した. SSH と Telnet の低対話型ハニーポットで知られる Cowrie [14] に 508 個の IP アドレスを割り当て, 2017 年 8 月から 2019 年 9 月までの約 3 年間に断続的に運用した. Perl スクリプトなどの ELF ファイルでないマルウェアは除外した.

収集した検体をアーキテクチャごとに分類したものを表 1 に示す. 収集した検体には 9 種類のアーキテクチャの検体が含まれ, 最も割合が多いアーキテクチャは Intel 80386 であった. また, 32 ビットの検体が 93.9%を占めた. 98.5%の検体が静的結合され, 78.5%の検体はシンボル情報が消去されていた. つまり, 78.5%の 2,148 検体においてライブラリ関数が明らかでない.

収集した検体のマルウェアファミリの割合を表 2 に示す. マルウェアファミリ名は, VirusTotal で調べた結果を AVClass [15] で特定した.

表 1　収集した検体のアーキテクチャごとの割合

アーキテクチャ	検体数	割合
Intel 80386	2,272	74.44%
MIPS 32-bit	408	13.37%
x86-64	187	6.13%
ARM 32-bit	103	3.37%
Renesas 32-bit	28	0.92%
Power PC 32-bit	18	0.59%
Motorola 32-bit	17	0.56%
SPARC 32-bit	14	0.46%
ARC 32-bit	5	0.16%

表 2　マルウェアファミリの割合

マルウェアファミリ名	検体数	割合
Mirai	2,492	81.65%
Gafgyt	475	15.56%
Xor DDoS	41	1.34%
Tsunami	17	0.56%
DDosSTF	2	0.07%
ChinaZ	1	0.03%
Lightaidra	1	0.03%
Setag	1	0.03%
Silex	1	0.03%
SSHgo	1	0.03%
XMRig	1	0.03%
分類不能	19	0.62%

最も割合が多いマルウェアファミリは Mirai であり, 全体の 81.65%であった. 分類できたマルウェアファミリの中で, Mirai, Gafgyt, Xor DDos, Tsunami, DDosSTF, ChinaZ, Setag が DDoS 攻撃を行うマルウェアであり, 収集したマルウェア全体の 99.2%を占めた. また, 19 検体は VirusTotal でのマルウェア名がベンダーによって異なるため, AVClass で分類できなかった.

収集した検体のうち 30.9% の検体はパッカーでパックされていた. パッカーはパターンマッチングによる特定を回避するためにマルウェアで用いられることが多い. パックされたマルウェアのプログラムコードは圧縮や暗号化処理が施されているため, ライブラリ関数を特定する前にアンパックする必要がある. Isawa らの論文 [16] によれば 2017 年 7 月から 2018 年 1 月に IoT 機器を標的にしたマルウェアをハニーポットで収集した結果, パックされた検体は全体の 1.76%であったことから, 最近の検体はパックされる割合が増加していることがわかる. 30.9%という割合は無視できない大きさであると考え, 本研究ではすべてのパックされた検体をアンパックして分析することにした. パックされた検体の 87.7%は UPX でパックされており, こ

れらの検体は UPX でアンパックした. 残りの 12.3%の検体は UPX 以外のパッカーを用いてパックされていたため，サンドボックスで検体を実行しアンパックされたタイミングで仮想メモリからアンパックされたプログラムコードを抽出した.

4. 関数の特定方法と特定精度の定義

4.1. YARA によるライブラリ関数の特定とルール生成

本研究では多くの検体がよく知られた既存のコンパイラでビルドされているとの仮定のもとで，よく知られた既存のツールチェインに含まれる C ライブラリのコードと検体のコードをパターンマッチングすることにより，ライブラリ関数を特定する．パターンマッチングにはマルウェア検知で定評のある YARA を使用する．マルウェア検知ではマルウェアのコード断片をルールとするが，本研究ではライブラリ関数を検知するため，ライブラリ関数のコード断片をルールとする.

静的結合でプログラムをビルドするとき，プログラムが使用する関数は libc.a などの静的ライブラリ内のオブジェクトファイルで定義された関数がプログラムに結合される. 結合するときリンカーはオブジェクトファイルの再配置テーブルに従って結合した関数の一部データを書き換える. つまり，再配置される部分を除いて，静的ライブラリ内のオブジェクトファイルの関数コードとプログラム内の関数コードは同じである.

そこで，静的ライブラリ内のすべてのオブジェクトファイルで定義されたすべての関数を順番に取り出し，関数ごとに YARA ルールのパターンを生成する．関数が ELF ファイルに結合されるときに再配置される領域は YARA のワイルドカードに置き換える．関数全体をパターンとして定義すると，YARA ルールファイルのサイズが大きくなるので，各関数のパターンサイズの上限を 200 バイトとする.

パターンが 7 バイト以下の YARA ルールは関数ではないコードを関数として誤検知することが非常に多いため，はじめにパターンが 7 バイトより大きな YARA ルールを使って 7 バイトより大きな関数を特定する．次に 7 バイトより大きな関数の領域を検体のコード領域から除外して，7 バイト以下の YARA ルールを使って 7 バイト以下の関数を特定する.

ワイルドカードを持つパターンは他の関数のパターンと同一になることや他の関数のパターンを包含することがある．このようなパターンには 1 つのパターンに複数の関数が含まれる．このようなパターンが関数のコードと一致した場合，複数の関数が該当し，それらの中から 1 つの関数を特定する必要がある．関数を特定するには関数の依存関係に基づいた結合順を読み解く必要があるが，本研究では Mirai などソースコードがあるマルウェアをビルドして得られた ELF ファイルの関数の結合順を手がかりにして関数の候補を絞り込む．この絞り込みにより，88%の関数を特定でき，残りは多くても 4 つまで絞り込むことができる.

4.2. 特定精度の定義

混同行列のクラス（真陽性，偽陰性，偽陽性，真陰性）を用いて YARA ルールによるライブラリ関数の特定精度を定義する．GCC でビルドしたプログラムは GNU リンカーによって関数は以下に示す順番で ELF ファイルに結合される.

(1) C ランタイム関数群
crt1.o, crti.o, crtbeginT.o で定義された関数
(2) プログラム定義関数群
(3) ライブラリ関数群
libc.a, libstdc++.a, libgcc.a などで定義された関数
(4) C ランタイム関数群
crtend.o, crtn.o で定義された関数

最初と最後の C ランタイム関数群と 3 番目のライブラリ関数群の関数をポジティブ関数と定義し，2 番目のプログラム定義関数群の関数をネガティブ関数と定義する．この定義により，関数の特定結果を以下の混同行列のクラスに分類できる.

- 真陽性：True positive（TP）
  C ランタイム・ライブラリ関数を正しく特定した
- 偽陰性：False negative（FN）
  C ランタイム・ライブラリ関数を特定できなかった
- 偽陽性：False positive（FP）
  特定した関数が C ランタイム･ライブラリ関数ではなかった
- 真陰性：True negative（TN）
  プログラム定義関数を正しく特定した

これらのクラスの混同行列から，ライブラリ関数の特定精度を定義する.

- 関数の特定精度 ：
  Accuracy = (TP + TN) / (TP + TN + FP + FN)
- 真陽性率（True positive rate）:
  ライブラリ関数を正しく分類した割合（カバー率）
  TPR = TP / (TP + FN)
- 真陰性率（True negative rate）:
  プログラム定義関数を正しく分類した割合
  TNR = TN / (TN + FP)

4.3. 特定精度の推定

3 章で述べたとおり，検体の 78.5%においてシンボル情報が取り除かれているため，ライブラリ関数を正しく特定したことを確認できない．そこで，4.2 節で述べたとおり YARA ルールにマッチした関数がライブラリ関数であるかを GNU リンカーによる関数の結合順序を手がかりにして真陽性，偽陰性，偽陽性，真陰性を推定する．

まず，軽量でマルチアーキテクチャに対応している逆アセンブリフレームワークである Capstone [17] を用いてマルウェア内の関数を特定する．次に，YARA ルールにマッチした関数をアドレス順に並び替えてから，最初の C ランタイム関数群以外の関数の中で最初にマッチした関数をライブラリ関数群の先頭関数と仮定する．最後の C ランタイム関数群の最初の関数の 1 つ前の関数をライブラリ関数の末尾関数と仮定する．つまり，ライブラリ関数の先頭関数群から末尾関数までの関数をライブラリ関数と仮定する．一方，プログラム定義関数群は最初の C ランタイム関数群の直後の関数からライブラリ関数群の先頭関数の直前の関数までと仮定する．この仮定では，正しい先頭関数を見逃したとき，本来の正しい先頭関数から誤って特定した先頭関数の間に含まれる関数がプログラム定義関数としてみなされ，特定精度が過大評価されることがある．この過大評価の問題は目視により確認して修正する．

4.4. 特定精度の有効性評価

ソースコードがあるマルウェア (BASHLITE [18], lightaidra [18], lizkebab [18], pnscan [18], Mirai[19]) をビルドした検体と，シンボル情報がある収集検体（Mirai が 30 検体，Gafgyt が 30 検体，Silex が 1 検体，Xor DDoS が 32 検体）の合計 93 検体を用いて特定精度の推定値が正しいかを評価した．検体のビルドには Aboriginal Linux 1.2.0 i586 で構築したツールチェインを使用した．評価では，すべてのライブラリ関数が YARA ルールとマッチしているかを目視で確認して，すべての検体のライブラリ関数の先頭関数と末尾関数が正しく特定できたことを確認した．ただし，関数の候補を 4 個以下まで絞れた場合には特定できたとした．調査の結果，すべての検体において関数の特定精度は 100%であったことから，ライブラリ関数の推定方法が有効であることを確認した．

静的結合されたプログラムのシンボル情報はプログラムの実行に不要であり，書き換えが可能である．そのため，解析を困難にする目的でシンボル情報が偽装されている可能性が考えられる．そこで，シンボル情報が残されていたすべての検体について関数名と関数のアドレスが偽装されていないかを確認したが，偽装された検体は存在しなかった．

5. ライブラリ関数の特定と分析

5.1. 分析する検体

すべての収集検体を調べる前に事前調査として Intel 80386 アーキテクチャの検体に限定して調査を行う．ライブラリ関数が動的結合されている検体はシンボル情報がなくても容易に関数を特定できるため，これらの検体を除外する．さらに，ライブラリ関数が静的結合されているがシンボル情報やデバッグ情報を含む検体はこれらの情報から関数を特定できるため，これらの検体も除外する．最終的に本研究はライブラリ関数が静的結合され，シンボル情報が削除された 2,148 検体を対象とする．

5.2. ツールチェインの特定

ツールチェインを構成する要素（コンパイラ、アセンブラやリンカーなど）の組み合わせが，ライブラリ関数のマンシコードを決める．このため，ツールチェインとライブラリ関数には密接な関係があり，ツールチェインの特定が多数のライブラリ関数の特定につながることがわかっている．そこで，本研究はツールチェインの特定を主軸にライブラリ関数の特定を行う．

5.2.1. ツールチェインの選定と YARA ルール生成

マルウェア開発者の多くはマルウェアの開発環境の構築に多くの時間をかけず，よく知られた既存のコンパイラとライブラリ（ツールチェイン）を使用してマルウェアをビルドしていると考えられる．本研究では，Mirai のインストールガイドに記載されている Firmware Linux シリーズとその後継である Aboriginal Linux，そのほかによく知られた既存のツールチェインの構築ツールの中から人気の 3 つのビルドツールを選定した．

ツールチェインを構築するツールとツールのリリース日を表 3 に示す．表 3 の構築ツールで構築されたツールチェインに含まれる C ライブラリと C ランタイムから YARA ルールを生成した．ツールチェインごとに生成した YARA ルールファイルは Github 上で公開している [6].

表 3　ツールチェインの構築ツール

構築ツール	リリース日
Firmware Linux 0.9.6 ~ 0.9.11 [20]	2009/04/02 ~ 2010/03/29
Aboriginal Linux 1.0.0 ~ 1.4.5 [21]	2010/09/04 ~ 2016/01/11
Buildroot 2018.02 ~ 2019.05 [22]	2018/04/10 ~ 2019/06/02
Yocto bitbake 1.40 [23]	2018/11/15
crosstool-NG 1.23.0 [24]	2017/04/20

5.2.3. ツールチェインの特定結果

関数の特定精度は混同行列の各クラスの個数に依存する．解析対象の検体はシンボル情報が削除されているため，

真陰性と偽陽性を数えることができない．このため，ツールチェインの特定では，関数の特定精度ではなく，ライブラリ関数のカバー率を使用する必要がある．本研究では，すべてのYARAルールを使って各検体のライブラリ関数の特定を試み，ライブラリ関数のカバー率が 100%のとき，ツールチェインを特定できたと判断することにした. なお，ツールチェインの特定は構築ツールに基づいて行った．

ライブラリ関数を特定した結果，すべての検体においてライブラリ関数のカバー率は 100%であり，すべてのツールチェインを特定した．すべての検体についてツールチェインの構築ツールを特定した結果を表 4 に示す．

表 4　特定できた構築ツールとツールチェイン

構築ツール（上段）とツールチェイン（下段）	検体数
Firmware Linux 0.9.6 i586 GCC 4.1.2, binutils 2.17, uClibc 0.9.30.1	2,093
Aboriginal Linux 1.4.4 i586 GCC 4.2.1, binutils 2.17, musl 1.1.12	28
Buildroot 2018.08 i686 GCC 7.3.0, binutils 2.31.1, uClibc-ng 1.0.30	21
Firmware Linux 0.9.6 i686 GCC 4.1.2, binutils 2.17, uClibc 0.9.30.1	4
不明 GCC 8.2.0, binutils 2.31.1, musl 1.1.19, i586	2

解析対象のすべての検体は，たったの 5 種類のツールチェインでビルドされていた．また，上位 4 種類のツールチェインが Web サイト上に公開されていたことから，多くのマルウェアが既存のツールチェインを使用していることが証明された．Firmware Linux 0.9.6 i586 のツールチェインでビルドされた検体が非常に多かった．この原因は，Mirai のインストールガイドが Firmware Linux 0.9.6 のツールチェインを使用していたためと考えられる．表 5 の最後のツールチェインは構築ツールを特定できなかった．

5.3. 使用言語とライブラリ

検体の使用言語とライブラリごとの割合を表 5 に示す．最も利用されているライブラリは uClibc であった．uClibc は Linux の組み込み機器向けのライブラリであり，GLIBC と比べて軽量で，移植性が高いライブラリの中でもよく知られたライブラリである．また，musl の検体が 1.4%含まれた．musl はライブラリ関数がアセンブリレベルで定義されており，静的結合されたプログラムに最適化されている．Aboriginal Linux ではバージョン 1.4.4 から musl がライブラリの標準として設定されていることからこのような結果になった．

表 6 検体の使用言語とライブラリの割合

言語	ライブラリ	検体数	割合
C	uClibc	2,118	98.6%
C	musl	30	1.4%

5.4. 構築ツールとマルウェアファミリ

表 4 の特定した構築ツール（ツールチェイン）の結果をもとに，構築ツールとマルウェアファミリの関係を分析した．その結果を表 6 に示す．表 6 の X は表 4 に示した構築ツール不明のツールチェインである．表 6 には表 2 のマルウェアファミリがいくつか含まれないが，これは 5.1 節で述べた分析検体に含まれなかったためである．表 6 から，Mirai 以外にも Gafgyt が Firmware Linux 0.9.6 を使用している．また，Mirai と Gafgyt には新しい構築ツールである buildroot 2018.08 を使用した検体があり，これら検体は 2019 年の 6 月頃に収集した検体であった．

表 7 構築ツールとマルウェアファミリの関係

構築ツール	Mirai	Gafgyt	分類不能
Firmware Linux 0.9.6 i586	2,081	11	1
Aboriginal Linux 1.4.4 i586	28	0	0
Buildroot 2018.08 i686	20	1	0
Firmaware Linux 0.9.6 i686	3	1	0
X	2	0	0

5.5. ライブラリ関数の名前リストによるマルウェア分類

マルウェアの分類手法である imphash [2]や impfuzzy [3]によれば，インポートされた API の名前リストを用いてマルウェアを分類できることがわかっている．また，イボットらの研究 [25] では関数などのシンボル情報を用いてマルウェアの検知の可能性も示している．

これらの研究を参考にして，検体のライブラリ関数の名前リストを用いてマルウェアの分類を行った．本研究で得られた関数名リストには`__uClibc_main` など C ライブラリ固有の内部関数が含まれ，これらの内部関数はツールチェインの種類やバージョンによって異なる．さらに関数が結合される順番も異なる．このように，関数名リストの内容はツールチェインの種類に強く影響を受ける．そこで，ツールチェインの影響を軽減するため，C ライブラリ固有の内部関数を関数リストから削除することにした．内部関数はインクルードファイルから参照できない関数であるので，本来はインクルードファイルから参照できない関数を削除するべきであるが，本研究では簡易的に調査を行うために関数名が`__uClibc_`で始まる関数や関数名の最後に`_internal` が付く関数などを経験的に内部関数であると判断できるものを除外した．また，パターンマッチングにより 1 つに絞れなかった関数は複数の関数名を連結し，1

つの関数として扱うことにした．最後に得られた関数名のリストを昇順に並び替えた．

検体の分類は関数リストのハッシュ値（SHA2）とファジーハッシュ値（TLSH [26]）のそれぞれによって行った．ファジーハッシュアルゴリズムは関数名リストのような小さいファイルの類似度を算出できないことがある．このため，小さなファイルにも対応したファジーハッシュアルゴリズムである TLSH を使用した．TLSH では，同一検体とみなす検体間の距離が 30（ssdeep 換算で類似度 90%）以下のとき，同一検体と分類する．SHA2 のハッシュ値による分類の結果を図 1 に示す．SHA2 のハッシュ値による分類では，263 種類の亜種に分類でき，TLSH のファジーハッシュ値による分類では，105 種類の亜種に分類できた．ただし，本研究では簡易の方法で関数名リストを作成したため，これらのリストには関数名だけでは判断できない内部関数が存在するので，実際の関数はさらに少なくなることが予想され，その結果，さらに亜種の個数は減少すると考えられる．

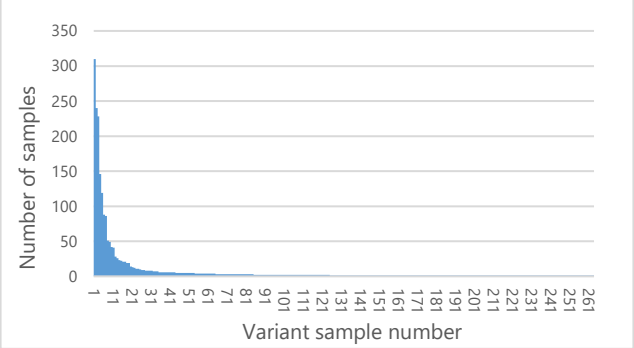

図 1　SHA2 のハッシュ値による分類結果

5.6. ライブラリ関数の個数

検体に静的結合されたライブラリ関数の数を分析した．検体に結合されたライブラリ関数の個数の階級とその階級に属する検体の数を図 2 に示す．図 2 から，多くの検体が 80 から 100 の階級に集中している．

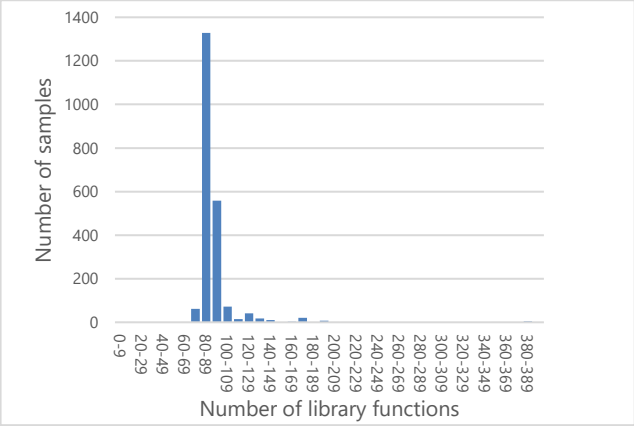

図 2　静的結合されたライブラリ関数の数と検体数

5.7. 検体ダウンロード日と構築ツールのリリース日

検体がダウンロードされた日と検体のビルドに使用されたツールチェイン（構築ツール）のリリース日の関係を調べた．調べた期間はハニーポットを連続稼働させていた 2018 年 9 月 29 日から 2019 年 9 月 14 日までである（2019 年 1 月 27 日から 2 月 10 日の約 2 週間を除く）．ダウンロード日とリリース日の相関係数は 0.019 であり，これらの間に相関はなかった．

ダウンロード日とリリースの散布図を図 2 に示す．リリース日が 2009 年の検体は Firmware Linux 0.9.6 でビルドされたものであり，年間を通してダウンロードされていた．2016 年の検体は Aboriginal Linux 1.4.4 でビルドされたものであり，2018 年 10 月から 12 月と 2019 年 3 月から 6 月の 2 つの期間にダウンロードされていた．2018 年の検体は Buildroot でビルドされたものであり，リリース日直後にダウンロードが観測され，2019 年 7 月までダウンロード数が増加していたが，それ以降はダウンロードされていない．

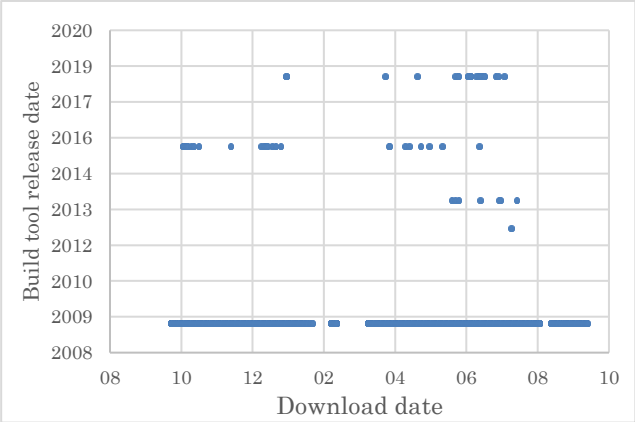

図 3　検体のダウンロード日と構築ツールリリース日

6.　結論と今後の課題

本研究では，シンボル情報が削除されライブラリ関数が静的結合された IoT マルウェアの解析を支援するために，IoT マルウェアに静的結合されたライブラリ関数の特定を行った．ハニーポットで収集した Intel 80386 アーキテクチャの 2,148 個のマルウェア検体で評価を行った結果，すべての検体についてすべてのライブラリ関数とツールチェインを特定した．すべての検体がたった 5 種類のツールチェインでビルドされ，97.4%の検体が Mirai のインストールガイドで紹介されていた Firmware Linux 0.9.6 のツールチェインでビルドされていることがわかった．検体が使用した言語を分析した結果，すべての検体が C 言語を使用していた．検体の 98.6%が uClibc，1.4%が musl の C ライブラリを使用していた．さらに，特定したライブラリ関数の名前リストに基づいて検体を分類した結果， SHA2 では約 263 種類の亜種が存在し，TLSH では約 105 種類の亜種が存在し

た．

今後の課題として，Intel 80386 以外のアーキテクチャの検体についてライブラリ関数を特定することが挙げられる．再配置の仕組みがアーキテクチャによって少し異なるため，アーキテクチャごとにパターンを生成する必要がある．また，今後の展開として，関数のトレースが挙げられる．静的結合されたライブラリ関数の呼び出しをトレースできれば，動的解析により関数の実引数の値を取得するなど，関数レベルの詳細な解析が可能になる．

参考文献

[1] Kaspersky. “IoT under fire: Kaspersky detects more than 100 million attacks on smart devices in H1 2019”. https://www.kaspersky.com/about/press-releases/2019_iot-under-fire-kaspersky-detects-more-than-100-million-attacks-on-smart-devices-in-h1-2019/, (参照 2020-02-19).

[2] Mandiant. “Tracking Malware with Import Hashing”. https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html/, (参照 2020-07-15).

[3] 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).

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

[5] Anna-senpai. “World’s largest net: Mirai botnet, client, echo loader, CNC source code release”. https://github.com/jgamblin/Mirai-Source-Code/blob/master/ForumPost.md/, (参照 2020-01-21).

[6] Akabane, S. Takeshi, O.. “stelftools”, https://github.com/shuakabane/stelftools/, (参照 2020-06-01).

[7] 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).

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

[9] Maximilian, v, T.. Library and Function Identification by Optimized Pattern Matching on Compressed Databases: A close to perfect identification of known code snippets. Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium, 2018, p. 1-12.

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

[11] 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.

[12] 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.

[13] 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.

[14] Michel, O.. “Cowrie”. https://www.cowrie.org/, (参照 2020-04-20).

[15] Marcos, S. and Richard, R. Platon, K. Juan, C.. Avclass: A tool for massive malware labeling. International Symposium on Research in Attacks, Intrusions, and Defenses, 2016, p. 230-253.

[16] Isawa, R. and Ban, T. Tie, Y. Yoshioka, K. Inoue, D.. Evaluating Disassembly-Code Based Similarity between IoT Malware Samples. Asia Joint Conference on Information Security, 2018, vol. 13, p. 89-94.

[17] Capstone. https://www.capstone-engine.org/, (参照 2020-04-30).

[18] Ding, F.. “iot-malware”. https://github.com/ifding/iot-malware/, (参照 2020-05-12).

[19] Gamblin, J.. “Mirai-Source-Code”. https://github.com/jgamblin/Mirai-Source-Code/, (参照 2020-05-12).

[20] Landley, R.. “Firmware Linux”. https://landley.net/code/firmware/old/, (参照 2020-03-12).

[21] Landley, R.. “Aboriginal Linux”. https://landley.net/aborigina/, (参照 2020-03-12).

[22] Korsgaard, P.. “Buildroot”. https://buildroot.org/, (参照 2020-03-12).

[23] Yocto Project. “Yocto”. https://www.yoctoproject.org/, (参照 2020-03-12).

[24] Day, R.: “crosstool-ng”. https://github.com/crosstool-ng/crosstool-ng/, (参照 2020-03-12).

[25] 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.

[26] Jonathan, O. Chun, C. Yanggui, Chen.. TLSH - A Locality Sensitive Hash. Cybercrime and Trustworthy Computing Workshop, 2013, vol. 4, p. 7-13.