Vol.2021-CSEC-93 No.13
Vol.2021-IOT-53 No.13
2021/5/13

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

赤羽 秀[1,*]　岡本 剛[1]

概要：IoT機器の増加とともに，IoT機器に感染するマルウェアが増加している．多くのIoTマルウェアにおいて，ライブラリ関数が静的結合され，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．我々の先行研究により，パターンマッチングによりIntel 80386のIoTマルウェアに静的結合されたすべてのライブラリ関数とツールチェインを特定できることがわかったが，Intel 80386以外のIoTマルウェアに対して我々の手法が有効であるかが明らかでなかった．そこで，我々の先行研究の手法をIntel 80386以外のIoTマルウェアに適用する手法を明らかにする．さらに，提案手法がIntel 80386以外のIoTマルウェアに対してツールチェインの特定に有効であるかを評価した．その結果，提案手法を用いて解析を行った3,991検体においてすべての検体のビルドに使用されたツールチェインを特定できることを確認した．検体のビルドに使用されたツールチェインは14種類であり，すべてのツールチェインがWebサイト上で公開されているものであった．Intel 80386以外の91.4%の検体がMiraiのインストールガイドで紹介されていたツールチェインでビルドされており，Intel 80386と同様の結果であった．本研究の成果をマルウェア対策コミュニティと共有するため，各検体のツールチェイン名と各検体に結合されたライブラリ関数の名前とそのアドレスのリストをGitHubに公開した．

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，IoT，パターンマッチング

Identification of toolchains used to build IoT malware with statically linked libraries

SHU AKABANE[1,*]　TAKESHI OKAMOTO[1]

Abstract: With the increase in IoT devices, there is an increase in malware infecting IoT devices. Much of IoT malware are built with static linking of library functions and all symbols such as function names and addresses are stripped, hindering function-level analysis. Our previous study showed that pattern matching can identify all library functions statically linked to Intel 80386 IoT malware and all toolchains used to build them, but it was not clear whether our method is effectively applied to IoT malware other than Intel 80386. Therefore, we propose a method to apply our previous method to IoT malware other than Intel 80386. In addition, we evaluated the effectiveness of our proposed method for identifying the toolchains used to build IoT malware other than Intel 80386. Our proposed method identified all toolchains used to build the 3,991 samples we collected. Only 14 toolchains had been used to build the samples, and the all toolchains are available on the Web sites. We found that 91.4% of the samples other than Intel 80386 were built with the toolchain described in the installation guide of Mirai, and this result was similar to that of Intel 80386. In order to share the results of this study with the anti-malware community, we published a list of the toolchain names of each sample and the names and addresses of the library functions linked to each sample on GitHub.

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

1. 序論

近年のIoT機器の普及に伴い，Linuxを利用するIoT機器を標的とした攻撃が増えている．SonicWall社の2020年9月までの年次調査では，景気後退によりマルウェアが減少傾向にあるのにもかかわらず，IoT機器を標的としたマルウェアは増加傾向にあると報告している [1]．増加するIoTマルウェアを効率よく解析するため，IoTマルウェアの解析を支援する技術開発が必要とされている．

IoTマルウェアの多くはELFファイルで提供されており，移植性の向上や解析の妨害を目的にライブラリ関数を静的に結合するマルウェアが多いことがわかっている [2-4]．さらに，マルウェア解析の重要な手掛かりとなるシンボル情報が削除されたIoTマルウェアが多数見つかっている [2-4]．シンボル情報が削除されると関数名や関数のアドレス情報が失われるため，関数レベルでの静的解析や動的解析が難しくなる．関数レベルでの解析を可能にするには関数の特定が必要であるが，静的結合された関数の名前を特定するにはマシンコードを読み解くなどの静的解析が必要であるため，高度な技術と多くの時間を要する．このような背景が原因であるかは明らかではないが，2018年にIEEE Symposium on Security & Privacyで発表されたLinuxマルウェアに関する調査研究[2]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．

1 神奈川工科大学
Kanagawa Institute of Technology
* s1822071@cco.kanagawa-it.ac.jp

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 [5] 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 [6-9] 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．パターンマッチングと類似度を組み合わせて関数を特定する方法 [10] は，はじめにパターンマッチングによりパターンに完全に一致する関数を特定し，次に一致しなかった関数について類似度に基づいて関数を特定する．他にも，いくつかの関数がリンクされたアドレスの相対距離からライブラリを特定する方法 [11] や，その方法に IDA F.L.I.R.T の高速な検索機能を組み合わせて関数を特定する方法 [4] などがある．前者の方法では，いくつかの関数のアドレスと関数名が事前に判明している必要があり，後者の方法は動的に結合されたライブラリ関数の特定を対象としており，実験では静的結合されたライブラリ関数は 17.2%しか特定していない．

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（システムやソフトウェアの開発に使用するコンパイラやアセンブラなどのツール群）を使っていることがわかっている [2]. さらに，Linux マルウェアは解析対策を積極的に行っていないことが知られており [2]，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，Web 上にバイナリで公開され，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た．そこで我々はその可能性を Intel 80386 アーキテクチャの IoT マルウェアについて調べた結果，我々の予想通り，検体の多くが Web 上に公開されたツールチェインを使用してビルドされていたことがわかった [12]. また，ツールチェインの特定により，すべてのライブラリ関数を特定することも可能であることがわかった（ただし，一部の関数は 4 つ程度の関数の中から 1 つの関数に絞り込む必要がある）．これらの結果から Intel 80386 以外のアーキテクチャの IoT マルウェアも Web 上に公開さ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．

本研究では，我々が Intel 80386 の IoT マルウェアを分析した手法 [12] を Intel 80386 以外のアーキテクチャの IoT マルウェアに適用する手法を提案する．さらに，提案手法が Intel 80386 以外の IoT マルウェアに対してもツールチェインの特定に有効であるかを明らかにする．ツールチェインの特定により，IoT マルウェアに静的結合されたライブラリ関数の 9 割程度を特定できることから [12]，ツールチェインの特定だけでもマルウェア解析に有用であると考える．我々の手法の有効性を明らかにするため，ハニーポットで独自に収集した IoT マルウェアの中から，動的結合された検体（67 個）と Go でビルドされた検体（1 個）とシンボル情報からツールチェインが特定できる glibc のすべての検体（45 個）を除外した，合計 3,991 個のマルウェア検体についてツールチェインの特定を試みた．本研究の成果は以下の通りである．

- Intel 80386 以外の検体に対してツールチェインを特定する方法を提案した．
- 我々の手法は Intel 80386 を含むすべての検体のツールチェインを特定できることを示した．
- すべての検体のツールチェインの特定に必要な YARA ルールを GitHub に公開した[13].
- すべての検体についてすべての関数のアドレスと関数名を逆アセンブラの Ghidra で利用できるように GitHub にそれらを公開した [13]．ただし，一部関数は関数名を特定していないため，複数の関数名の候補が含まれる．
- 特定したツールチェインは合計 14 種類あり，すべて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．
- Intel 80386 以外のアーキテクチャの検体の 91.4%が Firmware Linux 0.9.6 を使用してビルドされており，Intel 80386 と同様に Firmware Linux 0.9.6 の検体数が最も多かった．
- C ライブラリは 99.0%が uClibc，1.0%が musl を使用していた．

2. データセット

提案手法が Intel 80386 以外の IoT マルウェアに対してもツールチェイン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マルウェア検体に対してツールチェインの特定を行う. 検体の収集には，Cowrie [14] を使用し，2017 年 8 月から 2020 年 10 月まで断続的に運用して ELF ファイルのマルウェア検体を収集した. 収集した検体をアーキテクチャごとに分類した結果を表 1 に示す.

本研究は，ライブラリ関数が静的に結合された検体のツールチェインを特定することが目的であるため，収集した検体から動的結合された検体（67 個）を除外した．さらに glibc のすべての検体（45 個）はシンボル情報を保有し，このシンボル情報からツールチェイン名とすべてのライブラリ関数を特定できることから除外した. Go でビルドされた検体は 1 つしか存在しなかったため, 本研究では除外した. 最終的に残った検体は 3,991 個あり，これらを本研究のデータセットとする．データセットに含まれる検体をアーキテクチャ毎に分類した検体数を表 2 に示す.

収集した検体には 10 種類のアーキテクチャが含まれ，

表 1　収集したマルウェア検体のアーキテクチャごとの分類

アーキテクチャ	ライブラリ関数の結合方法		シンボル		パッカー		合計
	静的	動的	有	無	UPX	UPX 以外	
ARC	2	3	3	2	0	0	5
ARM	87	13	43	57	27	2	100
Intel 80386	3,082	17	937	2,162	1,141	193	3,099
MIPS	601	4	78	527	33	1	605
MIPS64	1	0	1	0	0	0	1
Motorola m68k	7	0	2	5	0	0	7
PowerPC	9	0	2	7	4	0	9
Renesas sh4	17	0	8	9	0	0	17
SPARC	4	0	3	1	0	0	4
x86-64	227	30	64	193	29	3	257

表 2　解析対象の検体のアーキテクチャごとの分類

アーキテクチャ	検体数	割合
Intel 80386	3,040	76.17%
MIPS 32-bit	601	15.06%
x86-64	225	5.64%
ARM 32-bit	85	2.13%
Renesas sh4	17	0.43%
Power PC 32-bit	9	0.23%
Motorola m68k	7	0.18%
SPARC 32-bit	4	0.10%
ARC 32-bit	2	0.05%
MIPS 64-bit	1	0.03%

データセット全体の 72.2%はシンボル情報が削除されていた. 収集した検体のうち 34.9% の検体はパッカーでパックされており，パターンマッチングを行うためには，アンパックする必要がある. パックされた検体の 86.1%は UPX でパックされていたため，これらの検体は UPX でアンパックした. 残りの 13.9%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.

収集した検体のマルウェアファミリの割合を表 3 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．最も割合が多いマルウェアファミリは Mirai であり，全体の 88.1%であった. 分類できたマルウェアファミリの中では DDoS 攻撃を行う検体が収集した検体全体の 99.8%を占めた．また，8 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 でマルウェアファミリを分類できなかった.

表 3　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	3,516	88.10%
Gafgyt	454	11.38%
Tsunami	12	0.30%
Silex	1	0.03%
分類不能	8	0.20%

3. ライブラリ関数の特定

ライブラリ関数のマシンコードは，ツールチェインを構成するライブラリやコマンド（コンパイラやアセンブラ，リンカーなど）の組み合わせによって変化する．そこで，本研究はツールチェイン毎にライブラリ関数のパターンを生成して，パターンマッチングによってライブラリ関数の特定を行う．パターンマッチングにはマルウェア解析で定評のある YARA を利用する．ライブラリ関数の特定手順を以下に示す.

(1) ライブラリ関数のパターン生成
(2) ライブラリ関数名の特定
(3) ツールチェインの特定

3.1 ライブラリ関数のパターン生成

事前準備として，ツールチェインが提供する libc.a などの静的ライブラリに含まれる各関数を順番に取り出し，各関数のオブジェクトのマシンコードから YARA ルールを生成する．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ため，各関数のパターンサイズの上限は 200 バイトとする．静的ライブラリ内の一部の命令は，再配置によるオペランド値の書き換えやリンカーによる命令の書き換えによって結合時に変化する．パターン生成では，再配置やリンカーによる命令の書き換

えに対応するために書き換えが発生するバイト列を YARA ルールのワイルドカードに置き換える．再配置のアドレスとサイズは再配置タイプ毎に定義され，1 バイトから 8 バイトまで様々なサイズが存在する．再配置タイプの定義はアーキテクチャごとに異なるため，アーキテクチャ毎に再配置タイプに合わせてワイルドカードに置き換える．

Intel 80386 の再配置タイプある R_386_TLS_GOTIE や，SPARC の再配置タイプである R_SPARC_WDISP30 は，再配置されるオペランドだけでなく，オペコードもリンカーによって書き換えられる．例えば，R_386_TLS_GOTIE はリンカーの最適化 [16] によって書き換えられる．その例を図 1 に示す．下線部の命令は再配置の対象であるため，結合前と結合後でオペランドの変化が確認できるが，ここではオペコードも変化している．

3.2 ライブラリ関数名の特定

ライブラリ関数名の特定では，検体に含まれるすべての関数のアドレスを取得してから，パターンマッチングにより取得したアドレスが関数のアドレスであることを検証し，同時に関数名を特定する．

3.2.1 関数のアドレスの取得

関数のアドレスを取得するための手法として，Nucleus [17] や ByteWeight [18] などの様々な手法が提案されている．しかし，多くの手法は Intel 80386 や x86-64 などの特定のアーキテクチャに特化した手法であり，特に解析妨害が施されているマルウェア定義関数のアドレスの特定が目的である．そのため，ライブラリ関数でないアドレスを過剰に取得することや，反対に 5 バイト以下の小さなライブラリ関数を見逃すことがある．我々が特定したい関数はライブラリ関数であり，我々の事前の研究ではライブラリ関数が難読化されていることはなかったため，上述の手法を使わずに単純な手法で関数のアドレスを取得することにする．その手法はアーキテクチャによって異なり，次の 3 つの手法のいずれかである．

- CALL 命令など関数を呼び出す命令のオペランドから関数のアドレスを取得する
- グローバルオフセットテーブル（GOT）のセクションから関数のアドレスを取得する
- 関数のコード領域の直後に設けられた領域から関数のアドレスを取得する

アーキテクチャごとに関数のアドレスを取得する方法をまとめたものを表 4 に示す．

関数を呼び出す命令のオペランドから関数のアドレスを取得する方法では，逆アセンブリフレームワークである Capstone [19] や，逆アセンブルコマンド objdump を用いて逆アセンブルを行い，各アーキテクチャの関数の呼び出しに使用する命令を探索し，その命令のオペランドを関数のアドレスとして取得する．図 2 に Intel 80386 における関数呼び出しの例を示す．図 2 の 3 行目で call 命令で関数を呼び出しているので，call 命令のオペランドに指定してされたアドレスを関数のアドレスとして取得する．

```
5  ...
   800004b  8d 04 1d 00 00 00 00   lea eax, [ebx]
6  8000052  e8 fc ff ff ff        call 0x8000053
   ...
                    ⇓                        ⇓
   ...
5  80529bf  65 a1 00 00 00 00      mov eax, dword gs:[0]
6  80529c5  81 e8 04 00 00 00      sub eax, 4
   ...
```

図 1　リンカーの最適化による書き換えの例

表 4　関数のアドレスの取得箇所

アーキテクチャ	アドレスの取得箇所
ARC	bl.d 命令のオペランド
ARM 32-bit	bl 命令のオペランド
MIPS	GOT セクション
MIPS64	GOT セクション
Motorola m68k	bsrl 命令のオペランド
PowerPC	bl 命令のオペランド
Renesas sh4	各関数のコード領域の直後の領域
SPARC	call 命令のオペランド
Intel 80386	call 命令のオペランド
x86-64	call 命令のオペランド

グローバルオフセットテーブル（GOT）とは，関数などがアドレスに依存することなく，どのアドレスにリンクされても正しく呼び出せるようアドレスの位置を記録するセクションであり，バイナリのリンク時や実行時にアドレスが記録される．MIPS の検体では，ライブラリ関数を静的に結合する場合であっても結合された関数のアドレスが GOT セクションに記録されるため，セクション情報をもとに GOT のアドレスとサイズを計算し，GOT セクション内に記録されているアドレスを順番に取り出し関数のアドレスとして取得する．ただし，GOT の先頭からいくつかは関数のアドレスではないアドレスが含まれることがあるが，その対処方法は 5 章の考察で述べる．図 3 に MIPS の GOT セクションの例を示す．GOT セクションの先頭から 4 バイトずつ関数のアドレスとして取得する．

sh4 アーキテクチャでは，各関数のコード領域の直後に設けられた領域に呼び出す関数のアドレスが記録されている．関数を呼び出す際は，関数のアドレスが記録されている領域を参照しレジスタに関数のアドレスをロードし，レジスタを指定して関数を呼び出す．図 4 に sh4 における関数呼び出しの例を示す．図 4 の 1 行目で r1 レジスタに指定したアドレス先に記録されている関数のアドレスをコピーしてから，6 行目の jmp 命令のオペランドに r1 レジスタを指定し，関数を呼び出している．9 行目には呼び出し先である random 関数のアドレスが記録されている．この 9 行

```
…   …          …              …
1   804a7af    3d 00 f0 ff ff cmp  eax, 0xfffff000
2   804a7b4    76 0c          jbe  0x804a7c2
3   804a7b6    e8 97 dc ff ff call 0x8048452
4   804a7bb    f7 db          neg  ebx
5   804a7bd    89 18          mov  DWORD PTR [eax], ebx
…   …          …              …
```

図 2　Intel 80386 検体の例　（kill 関数のコード断片）

GOT セクション	関数のアドレス
00 40 d5 20	0x40d520
00 40 d2 20	0x40d220
00 40 cc d0	0x40ccd0
00 46 40 60	0x464060
00 40 d3 f0	0x40d3f0

図 3　MIPS 検体の GOT セクションの断片の例

```
1   40cb74   03 d1       mov.l 0x40cb84, r1
2   40cb76   e6 2f       mov.l r14, @-r15
3   40cb78   f3 6e       mov   r15, r14
4   40cb7a   e3 6f       mov   r14, r15
5   40cb7c   f6 6e       mov.l @r15+, r14
6   40cb7e   2b 41       jmp @r1
7   40cb80   09 00       nop
8   40cb82   09 00       nop
9   40cb84   88 cb 40 00
```

図 4　sh4 検体の例（rand 関数のコード）

目のアドレスを関数のアドレスとして取得する．

3.2.2 関数名の特定

生成した関数のパターンを用いて，パターンマッチングによる関数の特定を行う．ただし，小さな関数のパターンは，関数ではないコード断片と誤って一致することが多いため，最初のパターンマッチングでは小さな関数を除外する．ここで，小さな関数とは X バイト以下の関数として定義する．この時点でパターンと一致したアドレスを X バイトを超える関数のアドレスとする．さらに，一致したアドレスを前項で取得したアドレス群から除外する．次にアドレス群に残った未特定関数のアドレスに対して小さな関数のパターン，具体的には X バイト以下のパターンを使って関数を特定する．

上述の X バイトの長さは長すぎると小さな関数を見逃すことになり，一方で短すぎると内部関数のコードを関数として誤って特定することになるため，最適な長さを決める必要がある．また，X バイトの長さは，アーキテクチャの命令セットに依存するため，アーキテクチャごとに最適な長さを決定することになる．そこで，いくつかの検体について X バイトの長さを 4 バイトから 12 バイトまで変更して最適な大きさを調べた．その結果，最適な大きさは Intel 80386 では 7 バイト，PowerPC では 5 バイト，m68k や sh4, SPARC では 4 バイト，それ以外のアーキテクチャでは 6 バイトであることがわかった．なお，X バイトの長さにワイルドカードの長さは含まれない．パターンマッチングによる関数名の特定では，一部の関数はライブラリ関数であることを特定したが，関数名に複数の候補があり特定でない場合がある．これは関数のコードが同一であるが関数名が異なることが原因で発生する．YARA ルールにおいて再配置のためのワイルドカードが原因で発生することもある．このようにライブラリ関数名を特定できないことがあるが，ライブラリ関数であることに間違いはなく，いくつかの候補を特定しているので，本稿では 1 つのライブラリ関数に対して 1 つ以上の関数名を特定した場合は，関数名を特定したと判断する（本稿ではツールチェインを特定することが目的であるため，複数の関数の候補から 1 つの関数に絞る作業を行わない）．さらに，検体に結合されたライブラリ関数の名前をすべて特定したとき，ツールチェインを特定したと判断する．これはツールチェインが異なればライブラリ関数が異なり，必ず特定できないライブラリ関数が存在するためである．

3.3 ツールチェインの特定

ツールチェインの特定では，すべてのライブラリ関数を特定できたとき，パターンマッチングに使用したパターンの生成元のツールチェインを検体のビルドに使用されたツールチェインとする．なお，ツールチェインの構成要素が他のツールチェインと同じであることが原因で複数のツールチェインにおいて，すべてのライブラリ関数を特定できる場合がある．そのような場合，本稿では複数のツールチェインの中から最も古いツールチェイン名のみを記載する．

4. ツールチェインの特定と分析

4.1 ツールチェインの選定

マルウェア開発者の多くはツールチェインを改造しないで，よく知られた既存の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，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 つのツールチェイン構築ツール（Buildroot, Yocto, Crosstool-NG）と，Mirai のインストールガイド [20] に記載されている Firmware Linux シリーズとその後継である Aboriginal Linux，さらに，ツールチェイン構築ツールの Buildroot でビルドされたツールチェインを公開しているプロジェクトの中から，2 つのプロジェクト（Synopsys, Bootlin）が提供するツールチェインを追加した合計 7 種類のツールチェインを選定した．

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 5 に示す．表 5 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 ランタイムからライブラリ関数のパターンである 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 [13].

表 5　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 [21]	2009/04/02 ~ 2010/03/29
Aboriginal Linux 1.0.0 ~ 1.4.5 [22]	2010/09/04 ~ 2016/01/11
Buildroot 2018.02 ~ 2019.05 [23]	2018/04/10 ~ 2019/06/02
Yocto BitBake 1.40 [24]	2018/11/15
Crosstool-NG 1.23.0 ~ 1.24.0 [25]	2017/04/19 ~ 2019/04/13
Buildroot	2017/10/31
(Synopsys prebuilt toolchain [26])	
Buildroot	2018/11
(Bootlin prebuilt toolchain [27])	

4.2 ツールチェインの特定結果

ツールチェインの特定を行った結果，すべての検体のツールチェインを特定した．その結果を表 6 に示す．すべての検体は，14 種類のいずれかの構築ツールで構築したツールチェインでビルドされていた．また，ツールチェインは特定できたが，バイナリは公開されていなかった Crosstool-NG を除く 13 種類のツールチェインが Web サイト上にバイナリで公開されていたことから，ほぼすべてのマルウェアが Web 上に公開されたバイナリのツールチェインを使用していたことが証明された．Firmware Linux 0.9.6 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．

4.3 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 7 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 1.0%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．

4.4 ツールチェインとマルウェアファミリ

全検体の構築ツール（ツールチェイン）の特定結果をもとに，3 つのマルウェアファミリの構築ツールの関係を分析した．結果を表 8 に示す．Intel 80386 以外のアーキテクチャの 867 検体が Firmware Linux 0.9.6 を使用してビルドされており，Intel 80386 と同様に Firmware Linux 0.9.6 の検体数が最も多かった．また，マルウェアファミリごとの検体数も Intel 80386 と同様に Firmware Linux 0.9.6 でビルドされた検体数が最も多かった．さらに，Intel 80386 以外のア

表 6　構築ツールとマルウェアファミリの関係

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	3,830
GCC 4.1.2, binutils 2.17, uClibc 0.9.30.1	
Aboriginal Linux 1.1.0	8
GCC 4.2.1, binutils 2.17, uClibc 0.9.32	
Aboriginal Linux 1.1.1	6
GCC 4.2.1, binutils 2.17, uClibc 0.9.32.1	
Aboriginal Linux 1.2.0	17
GCC 4.2.1, binutils 2.17, uClibc 0.9.33.2	
Aboriginal Linux 1.2.1	11
GCC 4.2.1, binutils 397a64b3, uClibc 0.9.33.2	
Aboriginal Linux 1.2.4	27
GCC 4.2.1, binutils 397a64b3, uClibc 0.9.33.2	
Aboriginal Linux 1.2.6	22
GCC 4.2.1, binutils 2.17, uClibc 0.9.33.2	
Aboriginal Linux 1.4.3	6
GCC 4.2.1, binutils 397a64b3, uClibc 0.9.33.2	
Aboriginal Linux 1.4.4	36
GCC 4.2.1, binutils 2.17, musl 1.1.12	
Buildroot 2018.08 i686 (Bootlin toolchain)	15
GCC 7.3.0, binutils 2.29.1, uClibc-ng 1.0.30	
Buildroot 2018.08 core2 (Bootlin toolchain)	7
GCC 7.3.0, binutils 2.29.1, uClibc-ng 1.0.30	
Buildroot 2018.08 core-i7 (Bootlin toolchain)	2
GCC 7.3.0, binutils 2.29.1, musl 1.1.19	
Buildroot 2018.08 arc700 (Synopsys toolchain)	2
GCC 7.1.1, binutils2.29 , uClibc-ng 1.0.26	
Crosstool-NG 1.24.0-rc1 i686	2
GCC 8.2.0, binutils 2.30, musl 1.1.19	

表 7　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	3,951	99.00%
C	musl	40	1.00%

表 8　構築ツールとマルウェアファミリの関係

構築ツール		マルウェアファミリ		
		Mirai	Gafgyt	Tsunami
Firmware Linux	Intel 80386	2,902	53	0
	Intel 80386 以外	489	366	12
Aboriginal Linux	Intel 80386	53	1	0
	Intel 80386 以外	45	33	0
Buildroot	Intel 80386	23	1	0
	Intel 80386 以外	4	0	0

ーキテクチャでも Intel 80386 と同様に Mirai の検体が多い結果となった．各アーキテクチャの検体について分析すると，ARM 32-bit の検体では，Firmware Linux よりも Aboriginal Linux を利用する検体が多かった．Aboriginal Linux では 1.2.0 からスレッドライブラリに NPTL を選択していることが開発者からアナウンスされているため，この恩恵を受けたいマルウェア開発者が Firmware Linux の後継にあたる Aboriginal Linux の比較的新しいバージョンを意図的に選択しているのではないかと考えられる．

5. 考察

MIPS の ELF ファイルでは関数のアドレスが GOT セクションに記述されているため，GOT セクションの先頭から終端までのアドレスを関数のアドレスとして抽出した．しかし，MIPS の GOT セクションの先頭には，内部関数 _dl_runtime_resolve のアドレスや，ELF ヘッダのアドレス，特定のオブジェクトへのアドレスなどが格納されており，すべての GOT セクションの値が関数のアドレスとは限らない．さらに特定のオブジェクトへのアドレスのエントリ数は検体ごとに異なることがある．本稿では GOT セクションのすべてのエントリが関数のアドレスと仮定して，ライブラリ関数を特定できなかったとき，手動で逆アセンブルしてライブラリ関数でないことを確認していた．本研究のデーセットには手動での確認が必要な検体が少なかったが，手動で確認する必要がある検体が多い場合にはこの処理を自動化する必要がある．

SPARC の検体では，memcpy 関数の内部関数に無名関数が存在することを確認している．無名関数は名前がないため，関数名の特定を行わなかったが，無名関数も特定する必要がある場合は，YARA ルールを生成するとき __leaf_noname_memcpy のような関数名を自動的に付ければ，関数を特定できる．

6. 結論と今後の課題

本研究では，我々がで提案した手法[12]が Intel 80386 以外のアーキテクチャの検体に対して有効であるかを評価するため，Intel 80386 とそれ以外の合計 10 種のアーキテクチャの検体についてツールチェインの特定を試みた．ハニーポットで収集した 3,991 個のマルウェア検体に対してツールチェインを特定した結果，すべての検体のビルドに使用されたツールチェインを特定した．すべての検体は，14 種類のツールチェインのいずれかでビルドされており，検体をアーキテクチャごとに集計すると検体は 1 種類から 7 種類のツールチェインのいずれかでビルドされていた．96.0%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おり，Intel 80386 以外の検体も 91.4%の検体が Firmware Linux 0.9.6 を使用してビルドされていた． これらの結果は Intel 80386 の検体と同様の結果である． C ライブラリは検体の 99.0%が uClibc, 1.0%が musl を使用していた．

今後の課題として，関数の候補の絞り込みが挙げられる．小さいライブラリ関数は他のライブラリ関数と同じ YARA ルールを持つことがあり，このような YARA ルールとマッチした場合，1 つの関数に対して，複数の関数名が候補となるため，これらの候補から正しい関数を特定する必要がある．関数の依存関係を手がかりにすることで，関数を絞り込めると考えられる．関数の結合順は結合される関数同士の依存関係に基づいて決定されるため，静的ライブラリ内の関数の依存関係を解析することで，関数の候補を絞り込むことが可能になる．さらなる課題として，Go や Rust などの新しい言語でビルドされた検体や LLVM ベースの Clang でビルドされた検体のライブラリ関数とツールチェインの特定が挙げられる．

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] イボット アリジャン. 大山 恵弘. IoT マルウェアの分類における画像化を用いた手法とシステムコール列を用いた手法の比較. 研究報告コンピュータセキュリティ(CSEC), 2021, p. 1-8.

[4] 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.

[5] 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).

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

[7] 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.

[8] 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.

[9] 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.

[10] Jacobson, ER. and Rosenblum, N. Miller, BP.. Labeling Library Functions in Stripped Binaries. Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, 2011, p. 1-8.

[11] Karlsruhe Institute for Technology CTF Team. “libc-database”. https://kitctf.de/tools/, (参照 2020-04-20).
[12] 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.
[13] Akabane, S. Okamoto, T.. “stelftools”, https://github.com/shuakabane/stelftools/, (参照 2020-06-01).
[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] Drepper, U.. “ELF Handling For Thread-Local Storage”, https://www.uclibc.org/docs/tls.pdf, (参照 2021-03-10)
[17] Andriesse, D. and Slowinska, A. Bos, H.. Compiler-Agnostic Function Detection in Binaries. IEEE European Symposium on Security and Privacy, 2017, p. 177-189.
[18] Bao, T. and Burket, J. Woo, M. Turner, R. Drumley, D.. ByteWeight: Learning to Recognize Functions in Binary Code. 23rd USENIX Security Symposium, 2014, p. 845-860.
[19] Capstone. https://www.capstone-engine.org/, (参照 2020-04-30).
[20] 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).
[21] Landley, R.. “Firmware Linux”. https://landley.net/code/firmware/old/, (参照 2020-03-12).
[22] Landley, R.. “Aboriginal Linux”. https://landley.net/aboriginal/, (参照 2020-03-12).
[23] Korsgaard, P.. “Buildroot”. https://buildroot.org/, (参照 2020-03-12).
[24] Yocto Project. “Yocto”. https://www.yoctoproject.org/, (参照 2020-03-12).
[25] Day, R.: “Crosstool-NG”. https://github.com/crosstool-ng/crosstool-ng/, (参照 2020-03-12).
[26] Synopsys.“toolchain”, https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain/, (参照 2021-03-20).
[27] Bootlin.“toolchains”, https://toolchains.bootlin.com/, (参照 2021-03-7).