Vol.2018-CSEC-81 No.2
Vol.2018-IOT-41 No.2
2018/5/17

Raspberry Pi環境における ステルス性の高い仮想マシン検出

大山 恵弘[1,a)]

概要：最近のマルウェアには Raspberry Pi を対象とするものがあり，Raspberry Pi 環境におけるマルウェアの挙動の解析が必要になっている．PC 環境を対象とするマルウェアには，仮想マシンなどの解析システムを示唆する機構を検出して実行終了などの解析回避活動を行うものがある．Raspberry Pi 環境を対象とするマルウェアも同様の活動を行う可能性があり，その脅威を理解することが必要である．本研究では，マルウェアが解析回避のために，自身の Raspberry Pi 環境が仮想マシンであるか実マシンであるかを性能情報のみから推定する方式を示す．その方式は，OS の基本操作の性能を整数演算の性能に対する相対値として表し，それを閾値と比較して推定する．その方式の特徴は，マルウェアが推定を試みていることを解析システムが特定しにくいことである．その方式に基づくプログラムを Raspberry Pi 2 の仮想マシンと実マシンの上で実行した結果，その方式によって高い精度での推定ができることがわかった．

Stealthy Virtual Machine Detection in Raspberry Pi Environments

Oyama Yoshihiro[1,a)]

Abstract: Some modern malware programs target Raspberry Pi and analysis of malware behavior in Raspberry Pi environments is needed. Some malware programs for PC environments detect analysis systems such as virtual machines and then execute analysis evasion activities such as execution termination. Malware programs for Raspberry Pi environments can also execute similar activities and understanding the threat of them is needed. In this study, we show a method with which malware programs estimate for analysis evasion whether their Raspberry Pi environment is a virtual machine or real machine based on performance information only. In this method, malware measures the performance of basic operating system operations relative to those of integer arithmetic operations, and estimates by comparing it with a threshold. The characteristic of the method is that analysis systems will have difficulty in determining that the malware program is attempting the estimation. We executed programs based on the method on a virtual machine and real machine of Raspberry Pi 2, and found that the method enabled accurate estimation.

1. はじめに

IoT デバイスを対象とするマルウェアが多数出現している．例えば 2016 年に大きな被害をもたらした Mirai は，IP カメラやデジタルビデオレコーダーを乗っ取ってボットネットを構成することが知られている [1]．また，トロイの木馬型マルウェアである Linux.MulDrop.14 は，Raspberry Pi を攻撃の標的に含み，感染した機器や PC をボットネット化して暗号通貨を採掘することが報告されている [8]．

[1] 筑波大学
[a)] oyama@cc.tsukuba.ac.jp

IoT デバイスを対象とするマルウェアの増加に伴い，それらのマルウェアを解析するシステムも必要となっている．一般に，動的解析においては，マルウェアを解析システム上で実際に実行することになる．動的解析においては，しばしば，ハイパバイザやエミュレータが作る仮想マシン上でマルウェアを実行する．例えば，様々な IoT デバイスを模擬するハニーポットである IoTPOT [17] は，QEMU が作る仮想マシン上でマルウェアを実行する．

洗練されたマルウェアは解析を回避するための処理（**解析回避処理**）を実行することが知られている．例えば，解析システムを検出すると自身の実行を終了したり，ダミー

処理を繰り返し実行したりする．解析回避処理は PC 向けのマルウェアの多くに組み込まれており，活発に研究されてきた [2, 4, 16, 18]．それらは IoT デバイスを対象とするマルウェアにとっても有効であり，IoT デバイス上でそれらを実行するマルウェアも報告されている [21]．

重要な解析回避処理の一つは仮想マシン検出である．マルウェアは自身が動作している環境の情報を収集し，それが実マシンか仮想マシンかを推定する．動的解析のための多くのサンドボックスは仮想マシンを利用して実装されているため，その環境が仮想マシンならば，自身がサンドボックス内で解析されている可能性は高くなる．仮想マシン検出は主に PC 環境を対象に研究されてきており，現状では IoT デバイスを対象とした仮想マシン検出の方法については深く理解されていない．特に，どのような処理によりどの程度正しく推定できるかは明らかではない．

本研究では，自身が動作している IoT デバイスが仮想マシンか実マシンかを推定するための方式を示す．現状では IoT デバイスが Raspberry Pi 2，OS が Raspbian (Linux)，CPU が Arm であることを仮定する．また，仮想マシンを動かすエミュレータとして QEMU を用いることを仮定する．今後，対象範囲を他の IoT デバイスや仮想化ソフトウェアにも拡げていく．本方式は，推定を試みていることを解析システムに検出されにくくしている（ステルス性を高くしている）点に特徴がある．それは性能情報のみを推定に用い，既存方式の多くが用いているファイルの存在やプロセス名などの資源の情報を用いない．一般に，資源の情報を収集するには，そのための API 関数を呼び出す必要がある．しかし，それらを呼び出すと，呼び出し情報を解析システムに収集され，解析回避の意図を解析者に疑われることになる．本方式では呼び出す API 関数の種類を極端に絞ることにより，仮想マシン検出の意図を解析者からできる限り隠している．本方式に基づくプログラムを Raspberry Pi 2 の仮想マシンと実マシンの上で実行したところ，そのプログラムは高い精度で仮想マシンの存在を正しく判定した．

2. マルウェアによる仮想マシン検出

仮想マシン検出のための代表的な手法を以下で述べる．

Artifact ベース手法: 仮想マシンや仮想マシン上で動くゲスト OS を示唆する資源の名前や内容を探し，もし検出できれば仮想マシンが動作していると判定する．例えば，ハードウェアデバイス，ファイル，レジストリに対して，既知の名前や内容との照合が行われる．特定の番号の I/O ポートの入出力結果や，CPUID 命令などの特定の命令の実行結果がハイパバイザの存在を示唆することもある．

Ordinariness ベース手法: 実行環境がどの程度「普通」であるかに基いて，それが解析用の環境であるかどうかを判定する．例えば，一定の時間内にマウスカーソルが動かないならば解析用の環境であると判定する．他にも，短い OS 稼働時間，小さいディスクサイズやメモリサイズ，少ないコア数などの情報が，解析環境を示唆する情報として用いられることがある．

性能ベース手法: 特定の処理の実行にかかる時間を測定し，それが実マシンで測定されるはずの値とは異なる場合に仮想マシンが存在すると判定する．例えば CPUID 命令は，仮想マシン内では実行に要する時間や CPU サイクル数が大きく増加することが多いため，実装によく用いられる．具体的には，CPUID 命令の実行前後に RDTSC 命令によって time stamp counter (TSC) の値を取得し，その値の差が所定の範囲の外にあれば仮想マシンが存在すると判定する．他には，API 関数の実行時間を測定する手法がある [3]．

仮想化不足ベース手法: 高速化や未実装などの理由で，ハイパバイザが実マシンの一部の動作を忠実にエミュレートしない場合がある．例えば，x86 用のハイパバイザにはデスクリプタテーブルに関する動作を十分に仮想化しないものがあり，デスクリプタテーブルのベースアドレスによって仮想マシンの存在を検出できる場合がある [19, 20]．

本研究で扱う方式では性能ベース手法を用いる．その理由は，仮想マシン検出を試みていることを解析システムや解析者に検出されにくくする点で有利であるからである．他の手法では特徴的な API 関数の呼び出しや CPU 命令の実行が必要となることが多く，その試みを検出されやすい．

性能ベース手法の実現においては，所定の処理の実行に要したサイクル数や時間の情報を取得することが必要になる．x86 環境を対象にした解析回避処理ではサイクル数の情報を RDTSC 命令によって簡便に得られる．RDTSC 命令はそれまでに実行された CPU サイクル数である TSC の値を返すものであり，ユーザ権限で実行可能である．動的解析システムには RDTSC 命令の実行を検出しないものもあり，そのようなシステムは，サイクル数の情報の取得という解析に役立つ手がかりを見逃すことになる．

Raspberry Pi が動作する Arm アーキテクチャの CPU では TSC に相当する値は特殊レジスタが保持しており，それを読み出すにはデフォルト設定の Raspbian ではカーネル権限が必要である．よって，ユーザプログラムがサイクル数の情報を得ることは難しく，gettimeofday や clock_gettime などのシステムコールで時刻情報を得る方が現実的である．また，ネットワーク通信によっても時刻情報は得られるが，通信を行うにはやはりシステムコールを実行する必要がある．しかし一般に，システムコールの捕捉は CPU 命令の実行の捕捉に比べて実現が容易でありオーバヘッドも小さいため，多くの解析システムはシステムコールを捕捉して解析する機能を有している．その結果，時刻情報を

表 1 Raspberry Pi 2 の実マシンの仕様

Machine model	Raspberry Pi 2 Model B Rev 1.2
Architecture	Armv7 rev 4
CPU	Arm Cortex-A53 (900 MHz)
Memory	1 GB
OS	Raspbian 8.0 (Linux 4.9.35-v7+)

表 2 Raspberry Pi 2 の仮想マシンの仕様

Machine model	Raspberry Pi 2 Model B
Architecture	Armv7 rev 1
CPU	Arm Cortex-A7 (900 MHz)
Memory	1 GB
OS	Raspbian 8.0 (Linux 4.9.35-v7+)

表 3 仮想マシンを動作させるホストマシン A の仕様

CPU	Intel Core i7-7700 (3.6 GHz)
Memory	32 GB
OS	Ubuntu 17.10 (Linux 4.13.0-36-generic)
Emulator	QEMU 2.10.1

得るためのシステムコールの実行は，サイクル数を得る命令の実行に比べて，解析システムや解析者に検出される可能性が高い．

3. 予備実験

Raspberry Pi 2 の仮想マシンと実マシンの性能に関する傾向を明らかにするために，基本演算や OS の基本操作の性能をそれらのマシン上で測定した．実験に用いた実マシンと仮想マシンの仕様をそれぞれ**表 1** と**表 2** に示す．本研究で仮想マシンを動作させるマシンの 1 つ（ホストマシン A）の仕様を**表 3** に示す．

仮想マシンと実マシンのハードウェアの型番は厳密には同一ではない．しかし，どちらも CPU アーキテクチャは Armv7 (32 bit) であり，それらの仕様は極めて近い．実マシンと仕様がわずかに異なる仮想マシンを用いた理由は，実験時に QEMU がサポートしていた Raspberry Pi の仮想マシンが Model B（無印）のもののみだったためである．仮想マシンのハードウェアの仕様は，QEMU のコマンド qemu-system-arm に，Raspberry Pi 2 Model B を表すオプション `-M raspi2` を与えて指定した．

まず，UnixBench ベンチマークの性能を測定した．使用した UnixBench のバージョンは 5.1.3 である．実マシンのコア数，仮想マシンの仮想コア数と物理コア数はどれも 4 である．結果を**図 1** に示す．デフォルトでは 1 並列と 4 並列でベンチマークが実行されるが，図は 4 並列の結果を示している．すべてのサブベンチマークで実マシンでの性能が仮想マシンでの性能を上回ったが，性能比はサブベンチマークによって大きく異なった．整数演算の性能を測定する Dhrystone 2 では仮想マシンでの相対性能は約

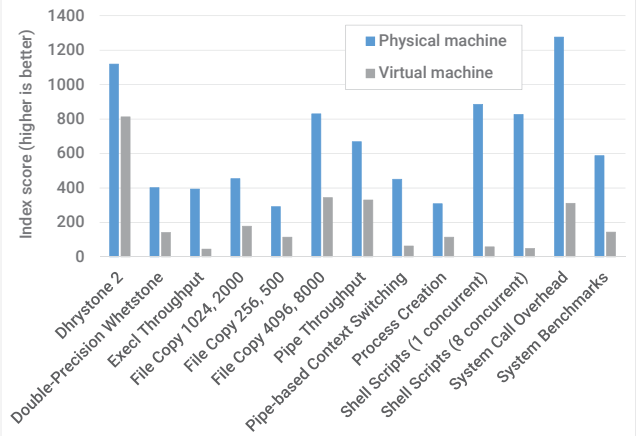

図 1 UnixBench の性能測定結果

0.73 であり，互いの性能は近かった．一方，相対性能が 0.2 以下であるサブベンチマークもあった．それらは，Execl throughput, Pipe-based Context Switching, Shell Scripts (1 concurrent), Shell Scripts (8 concurrent) の 4 つである．これらのサブベンチマークは，プロセス間通信，プロセス切り替え，プロセスの生成と終了，プロセスイメージの置き換えなどの，プロセスに関する処理を多く実行する．この結果から「仮想マシン上ではプロセスに関する処理の性能が他の処理の性能に比べて大きく低下する」という傾向があることが予測できる．実際，プロセスに関する処理の多くはページテーブルの切り替えや大幅な更新を伴うため，ソフトウェアで MMU をエミュレートする QEMU のようなシステムでそれらの性能が低下することは自然である．

次に，多くのページフォルトを発生させるための小さいプログラムを著者が記述し，その性能を測定した．このようなプログラムの性能を測定した理由は，ソフトウェアで MMU をエミュレートするシステム上ではページフォルトハンドリングも遅くなりやすいからである．そのプログラムは 800 MB のメモリ領域を確保し，その領域に含まれる各ページの先頭ワードのデータのみを 1 回ずつ読み出し，全ページからデータを読み出したらその領域を解放する，という一連の処理を繰り返す．800 MB は Raspberry Pi 2 上で確保できる限界に近いサイズである．各ページへのアクセスの際には，その仮想ページに物理ページが割り当てられていないことによるページフォルトが発生する可能性が高い．繰り返し回数を 10 回としてこのプログラムの実行時間を測定したところ，仮想マシンと実マシンでの実行時間比は 12 倍を超えた．この結果から「仮想マシン上ではページフォルトハンドリングの性能が他の処理の性能に比べて大きく低下する」という傾向があることが予測できる．

以上の観測から，本研究では仮想マシンと実マシンの間に以下の性質が成立すると仮定して方式を設計した．

- 整数演算の性能比は小さい
- プロセスに関する処理の性能比は大きい

- ページフォルトハンドリングの性能比は大きい

QEMU ではないシステムの上で動く仮想マシンでは，これらの性質は QEMU におけるほど顕著には表れない可能性がある．そのような環境で仮想マシンか実マシンかを推定する方式については本研究では扱わず，今後の課題とする．

4. 方式

4.1 概要

本研究で評価する方式（**評価対象方式**）は，**一定量の整数演算を実行する間に OS の基本操作を何回実行できるかを測定し，その回数を閾値と比較する**というものである．具体的には，この方式では以下の (1) と (2) の実行主体を並行に実行する．

(1) OS の基本操作を繰り返し実行するプロセス（またはスレッド）群
(2) 整数演算を実行するプロセス（またはスレッド）

(2) は整数演算を完了したら，シグナルや共有メモリを通じて (1) の中の 1 つに通知を送る．通知を受けた実行主体は，それまでに実行できている OS の基本操作の回数と閾値の比較結果に基づいて，仮想マシンの存在を推定する．上記の回数を以降では**スコア**と呼ぶ．

本研究で対象とした OS の基本操作を以下に示す．これらについては 4.2 節で詳しく述べる．

Pipe 2 つのプロセス間でのパイプによる通信
Fork プロセスの生成
Exec 実行可能プログラムのロードと実行
Forkexec プロセスの生成および，子プロセスによる実行可能プログラムのロードと実行
Pagefault ページフォルト

本方式で，OS の基本操作を所定の時間だけ実行するのではなく，一定量の整数演算を行う間だけ実行する最大の理由は，時刻取得関数の呼び出しの回避である．2 節で述べたように，特定の処理の実行に要する時間の測定は代表的な解析回避処理の一つであり，時刻の取得は「怪しい処理」として解析システムに検出される可能性がある．本方式では，時刻の取得のためのライブラリ関数も CPU 命令も実行されない．なお，時刻を取得しても良いならば，Raspberry Pi 2 環境を仮定しての仮想マシン検出は単純な問題となる．事前にマルウェア作者が Raspberry Pi 2 の実マシン上で所定のプログラムの実行時間を得ておき，判定したい環境上でそのプログラムの実行時間を測定すれば，高い精度で実マシンか仮想マシンかを判定できる．

別の理由は，仮想マシン検出を試みる環境内で負荷が高いプログラムが動いている場合におけるスコアの補正である．この場合にはそのプログラムの影響によって，単位時間あたりに実行できる OS の基本操作の回数が減る．また，低下の割合は必ずしも同じではないものの，整数演算を実行できる回数も同様に減る．よって，一定の時間内での実行回数ではなく，一定量の整数演算を行う間の実行回数を用いることにより，そのプログラムが推定に与える負の影響を縮小できる可能性がある．もし性能が低下する割合が整数演算と OS の基本操作とで等しいならば，そのプログラムによる影響を最小化することができる．

4.2 実装の詳細

4.2.1 Pipe

Pipe は，UnixBench の Pipe-based Context Switching サブベンチマークとほぼ同じ実装により，パイプによるプロセス間でのデータの送受信を繰り返し実行する．具体的には，まずプロセスを生成し，親プロセス P1 と子プロセス P2 の間に，双方向でデータを通信できるパイプの組を作る．P1 と P2 はこの組を用いて，1 ワードのデータを「ピンポン」する（受信用のパイプからデータを受信したら送信用のパイプにデータを送信する処理を繰り返す）．P1 は「ピンポン」の開始前に，一定量の整数演算を実行するスレッドを生成する．その子スレッドは整数演算の実行を完了したら共有メモリを通じて親スレッドにそれを通知する．親スレッドは通信が一往復するごとにその共有メモリを読んで通知の有無を調べる．通知を受けるまでにデータを往復送受信できた回数をスコアとする．

4.2.2 Fork

Fork は，プロセスを fork し，子プロセスの終了を waitpid システムコールで待つ処理を繰り返し実行する．子プロセスは実行開始後すぐに exit 関数を呼び出して終了する．別のプロセスが整数演算を実行し，その完了をシグナルで通知する．生成したプロセスの数をスコアとする．

4.2.3 Exec

Exec は，UnixBench の Execl Throughput サブベンチマークとほぼ同じ実装により，execl 関数を繰り返し実行する．このプログラムは，自身のプログラムファイルを第一引数に与えて execl を実行する．execl を実行した回数はプログラムの引数に保持する．execl の実行の際には，自分が受け取った値に 1 を足した値を引数に与える．別のプロセスが整数演算を実行し，その完了をシグナルで通知する．execl の実行回数をスコアとする．

4.2.4 Forkexec

Forkexec は，プロセスを fork し，子プロセスの終了を waitpid システムコールで待つ処理を繰り返し実行する．子プロセスは，何もしないで終了する（':' と同じ動作をする）プログラムを execl システムコールによって実行する．別のプロセスが整数演算を実行し，その完了をシグナルで通知する．生成したプロセスの数をスコアとする．

4.2.5 Pagefault

Pagefault は，3 節で述べたプログラムと同様に，大きいメモリ領域を確保し，各ページの先頭ワードのデータを順番に読み出し，その領域を解放する処理を繰り返す．別

```
uint32_t load(uint32_t n)
{
  int i, j, sum = 0;
  for (i = 0; i < n; i++) {
    for (j = 0; j < 1000000; j++) {
      sum += j;
    }
  }
  return sum;
}
```

図 2 整数演算を実行する関数

表 4 実マシンで測定して得られたスコアと仮想マシン検出のためのスコアの閾値

	Pipe	Exec	Fork	Forkexec	Pagefault
スコア	1046411	17836	30509	11304	18995660
閾値	523205	8918	15254	5652	9497830

のスレッドが整数演算を実行し，その完了を共有メモリによって通知する．ページを読み出しているスレッドは，一定数のページを読み出すごとに共有メモリを検査する．データを読み出したページの数をスコアとする．

4.2.6 整数演算

評価対象方式のプログラムは整数演算として**図 2** の関数を実行する．このプログラムは二重ループの中では整数演算や分岐命令などの CPU 命令を実行し続け，メモリには全くアクセスせず，関数も呼び出さない．

4.2.7 閾値

Raspberry Pi 2 の実マシン上で測定して得られたスコアを 2 で割った数を，仮想マシン検出のためのスコアの閾値として用いる．そのスコアと閾値を**表 4** に示す．スコアがこれらの閾値を下回ることは，測定された性能が，実マシンで得られるはずの性能の半分に達していないことを意味する．その場合には実行環境は仮想マシンであると判定し，そうでない場合には実マシンであると判定する．

5. 実験結果

実行環境が仮想マシンか実マシンかを評価対象方式を用いて判定するプログラム（**判定プログラム**）を開発し，複数の Raspberry Pi 2 環境において判定プログラムによる判定結果を得る実験を行った．この実験は，評価対象方式の判定能力を明らかにすることを目的としている．以下の環境で判定プログラムを実行した．

(1) 表 1 の実マシン
(2) 表 1 の実マシン（ただし CPU 資源を消費し続けるプロセスを他に 4 つ実行）
(3) ホストマシン A（表 3）上で実行した表 2 の仮想マシン
(4) ホストマシン B（**表 5**）上で実行した表 2 の仮想マシン
(5) ホストマシン C（**表 6**）上で実行した表 2 の仮想マシン（仮想マシンを実行する物理コアの指定は無し）

表 5 仮想マシンを動作させるホストマシン B の仕様

CPU	Intel Core i5-4570 (3.2 GHz)
Memory	8 GB
OS	Ubuntu 18.04 (Linux 4.13.0-25-generic)
Emulator	QEMU 2.11.1

表 6 仮想マシンを動作させるホストマシン C の仕様

CPU	Intel Core i5-560M (2.67 GHz)
Memory	4 GB
OS	Ubuntu 17.10 (Linux 4.13.0-37-generic)
Emulator	QEMU 2.10.1

(6) ホストマシン C（表 6）上で実行した表 2 の仮想マシン（4 物理コアのうち 2 物理コアのみの上で QEMU と仮想マシンを実行）

(2) の環境では，他のプログラムが高い負荷をかけている状態を作っている．実世界で運用されている Raspberry Pi の環境では，マルウェアと並行にアプリケーションが走っていることも多い．例えば Web サーバや，センサデータを解析する計算プログラムが動いていることがある．その場合，そのアプリケーションによる CPU サイクルやメモリなどの資源の消費が判定プログラムの性能に影響を与える可能性がある．

(6) の環境では，ホストマシンの CPU 資源が貧弱である状態を作っている．物理コアの割り当ては，QEMU が実行される物理コアを taskset コマンドで限定することによって実現した．仮想コア数は 4 のままである．

実験結果を**図 3** に示す．本稿で示す全ての測定値は 5 回の実行での平均である．グラフの縦軸は，閾値を 1 として正規化した相対スコアを表している．上で述べた (1) から (6) の環境は，グラフではそれぞれ Physical, Physical (extra load), VM on host A, VM on host B, VM on host C (4 cores), VM on host C (2 cores) に対応している．

判定プログラムは相対スコアが 1.0 以上ならば実マシン，1.0 未満ならば仮想マシンと判定するが，多くの場合において正しい判定を行った．判定を誤ったのは，Physical (extra load) の環境で Fork と Forkexec の基本操作を用いた場合のみである．これらの基本操作は正しい判定結果を与える場合も多いが，CPU 負荷が高いプログラムが動いている実マシンを仮想マシンであると判定する可能性がある．他の基本操作では実マシンでの性能と仮想マシンでの性能に大きい差がみられ，どの性能も閾値まで大きな距離があった．これらの基本操作を利用すれば，CPU 負荷が高いプログラムが動いている実マシンの環境や，測定される性能が少々揺らぐ環境でも正しい判定ができる可能性が高い．また，ホストマシンの性能が上がるほど，仮想マシン上で測定されるスコアも上がる傾向が観測された．ただし，最も性能が高いホストマシン A 上でのスコアでさえも，実マシン上でのスコアより大幅に低く，正しい判定が可能である．

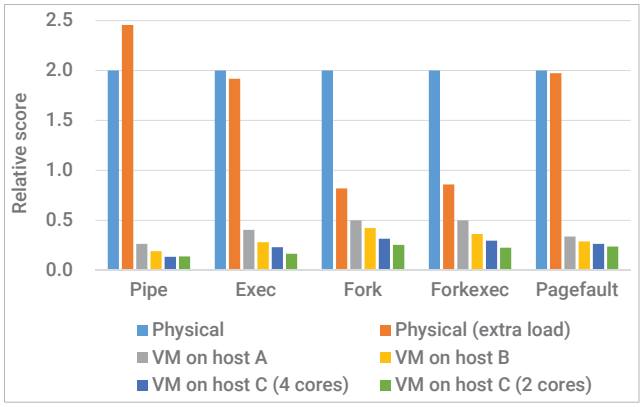

図 3 閾値を 1 として正規化したスコア

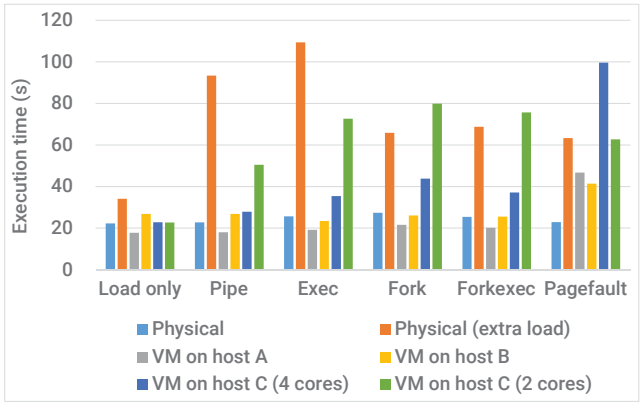

図 4 整数演算の実行時間

プロセス 1（PID: 28670）

```
pipe([3, 4])                                = 0
pipe([5, 6])                                = 0
# communication-peer process
clone(...)                                  = 28677
close(3)                                    = 0
close(6)                                    = 0
mmap2(NULL, 8388608, ...,
          ...|MAP_ANONYMOUS|..., -1, 0) = 0x7658f000
brk(0)                                      = 0xb99000
brk(0xbba000)                               = 0xbba000
mprotect(0x7658f000, 4096, PROT_NONE)       = 0
# computation thread
clone(...)                                  = 28678
write(4, "\0\0\0\0", 4)                     = 4
read(5, "\0\0\0\0", 4)                      = 4
...
write(4, "\267%\1\0", 4)                    = 4
read(5, "\267%\1\0", 4)                     = 4
kill(28677, SIGKILL)                        = 0
--- SIGCHLD { ..., si_pid=28677, ... } ---
wait4(28677, ...) = 28677
```

プロセス 2（PID: 28677）

```
set_robust_list(0x76fde280, 12)             = 0
close(4)                                    = 0
close(5)                                    = 0
read(3, "\0\0\0\0", 4)                      = 4
write(6, "\0\0\0\0", 4)                     = 4
...
read(3, "\267%\1\0", 4)                     = 4
write(6, "\267%\1\0", 4)                    = 4
read(3,  <unfinished ...>
+++ killed by SIGKILL +++
```

図 5 システムコールとシグナルの列（Pipe）

判定プログラムが整数演算の実行に要した時間を**図 4** に示す．それらの時間は環境によって大きくばらついている．特に，負荷を加えた (2) の環境では実行時間が大幅に伸びている．この伸びは，OS の基本操作を実行する時間も伸ばすことにつながり，ひいてはその環境で基本操作のスコアを上げて閾値から遠ざけることにつながっている．

6. 議論

6.1 デバイスの特定

本研究では，実行環境が Raspberry Pi 2 の実マシンまたは仮想マシンであることを，攻撃者が事前に得た情報やデバイス fingerprinting によって把握していると仮定している．それを把握するまでの部分は本研究の範囲外としているが，少なくともどれかの種類の Raspberry Pi であることをマルウェアが特別な処理なしに把握できているケースは稀ではないと考える．それは，多くのマルウェアは Raspbian のデフォルト値である pi というユーザ ID と raspberry というパスワードでログインを試み，それらによるログインの成功は，対象のマシンが Raspberry Pi であることを強く示唆するためである．本研究は，把握した後に解くべき問題に焦点を合わせている．

6.2 仮想マシン検出の意図の隠蔽

評価対象方式の処理を仮想マシン検出であると解析システムが判定する可能性を考える．判定プログラムを実マシン上で strace コマンドによって追跡しながら実行し，発行されたシステムコールと受信されたシグナルの列を取得した．それらのうち，Pipe，Forkexec，Pagefault の実行で取得された列をそれぞれ**図 5**，**図 6**，**図 7** に示す．なお，Exec と Fork が実行する処理の大半は Forkexec が実行する処理に含まれるため，それらについての記述は省く．

Pipe，Forkexec，Pagefault の実行中に発行されたシステムコールによる処理の説明を以下に示す．

Pipe パイプの生成，読み書き，クローズ，プロセスおよびスレッドの生成と待機，シグナルの送信，robust futex list の登録，メモリの確保と保護

Forkexec プロセスおよびスレッドの生成，待機，終了，robust futex list の登録，プログラムファイルの実行

Pagefault スレッドの生成と待機，メモリの確保と解放

これらは，多くの善良なプログラムも実行する一般的な処理であり，解析システムがこれらを解析回避処理だと判定する可能性は低いと考える．ただ，パイプによる多数回のデータ送受信や，何もせずに終了するプロセスの大量生成に対しては，その意図が不明と判定されて詳細な調査の対

```
親プロセス（PID: 28670）
# computation thread
clone(...)                                = 11252
# execve processes
clone(...)                                = 11253
wait4(11253, ...)                         = 11253
--- SIGCHLD { ..., si_pid=11253, ... } ---
clone(...)                                = 11254
wait4(11254, ...)                         = 11254
--- SIGCHLD { ..., si_pid=11254, ... } ---
...
clone(...)                                = 14171
wait4(14171, ...)                         = 14171
--- SIGCHLD { ..., si_pid=14171, ... } ---
```

```
子プロセス（PID: 11253, 11254, ..., 14171）
set_robust_list(...)                      = 0
execve("./nop", ["nop"], [/* 40 vars */]) = 0
...
（ここで main 関数の呼び出し）
exit_group(0)                             = ?
+++ exited with 0 +++
```

図 6 システムコールとシグナルの列（Forkexec）

```
# computation thread
clone(...)                                = 14172
mmap2(NULL, 838864896, ...,
              ...|MAP_ANONYMOUS, -1, 0) = 0x4458e000
munmap(0x4458e000, 838864896)             = 0
...
mmap2(NULL, 838864896, ...,
              ...|MAP_ANONYMOUS, -1, 0) = 0x4458e000
munmap(0x4458e000, 838864896)             = 0
mmap2(NULL, 838864896, ...,
              ...|MAP_ANONYMOUS, -1, 0) = 0x4458e000
# thread join
futex(0x76d8e4c8, FUTEX_WAIT, 14172, NULL) = 0
```

図 7 システムコールとシグナルの列（Pagefault）

象にされる可能性はあると推測する．しかし，その場合でも，それらは時間を稼いだり解析の手間を増やすためのダミー処理であると判定される可能性が高く，仮想マシン検出であると判定される可能性は低いと推測する．

CPU 使用率やページフォルト数を監視する解析システムは，評価対象方式がもたらすそれらの数値の著しい上昇を，注意すべき情報として解析者に伝える可能性がある．しかし，やはり解析者がそれらを仮想マシン検出によるものであると判定する可能性は低く，事実上のスリープ [13]，ファイル暗号化，コインマイニングなどの他の処理によるものであると判定する可能性が高いと推測する．

評価対象方式を意識した解析システムであれば，その方式の実行を検出することは可能である．ただし，その場合でも，実行される処理はそれほど特徴的ではなく，似た挙動を示すような他の目的の処理は他のアプリケーションにも存在するので，false negative や false positive を無くすことは難しい．また，実行を長時間停滞させるコードをスキップする処理が組み込まれた解析システム [13] は，評価対象方式の処理をスキップすることができる可能性があるが，同じ理由で false negative や false positive を無くすことは難しい．

7. 関連研究

PC 向けの仮想マシン検出に関しては多くの研究やシステムがある．Pafish [18] はサンドボックスや解析システムを検出するための多種の処理を実行するツールである．Pafish はファイル名やレジストリ名の検査に加えて，性能測定による仮想マシン検出も実行する．それには，CPUID 命令の実行に要するサイクル数を RDTSC 命令で取得する処理が含まれる．Franklin ら [9] は本研究と同じくベンチマークプログラムの性能によって仮想マシンの有無を推定する方式を示している．時間情報は RDTSC 命令で取得している．Kedrowitsch ら [11] は，コンテナ，ハイパバイザ，エミュレータを含む様々な仮想化機構上に構築された実行環境で，よく知られた複数の仮想化機構検出方式を評価している．CPUID 命令の実行に要するサイクル数を RDTSC 命令で取得する方式も評価しており，QEMU 上では実マシン上に比べてサイクル数のばらつきが大きいことなどを報告している．宮本らの研究 [22] では，RDTSC 命令そのものの実行に要するサイクル数が仮想マシン上と実マシン上で変わることを利用して仮想マシンを検出している．RDTSC 命令や CPUID 命令を用いてサンドボックスを検出する方法は多くの環境で有効であると同時に，広く知られている．実際，著名なマルウェア解析ツールである YARA にも，それらの命令をアンチデバッグとして検出するルールが存在する．よって，解析システムがそれらの命令の実行を解析回避処理であるとして検出する可能性は低くないと考える．また，通常の Raspberry Pi 環境では，ユーザプログラムはサイクル数情報を得る CPU 命令を実行できないため，この方法は使用できない．逆に，評価対象方式は Raspberry Pi 環境以外の広範な環境で使用できるため，PC 環境でも有効である可能性はある．その有効性については，さらなる実験によって今後明らかにする必要がある．

マルウェアにできるだけ検出させないことを意識して設計された解析システムが多数存在する [6, 7, 10, 12, 14, 15]．これらの研究は解析回避処理に対する解析システムのステルス性を扱っているが，本研究は解析システムに対する解析回避処理のステルス性を扱っている．

Cozzi ら [5] は Linux マルウェア 10548 検体を分析し，その挙動を明らかにしている．彼らの論文では仮想マシン検出を含む解析回避処理の実態が説明されているが，性能ベースの仮想マシン検出については全く言及されていない．

8. まとめと今後の課題

Raspberry Pi 環境において，その環境が仮想マシンか

実マシンかをプログラムの性能情報のみから判定する方式を示した．実験では，パイプによるデータの送受信，execl 関数の実行，ページフォルトを発生させるメモリアクセスの性能を用いた場合に，用意した全ての環境で正しい判定ができたことを示した．その方式は，解析回避処理であると疑われやすいシステムコールを実行しないという（マルウェア作者側にとっての）利点を有している．

今後の課題を以下に述べる．第一に，仮想マシンと実マシンで性能の傾向に違いが生じる理由を詳細に調査し，評価対象方式による仮想マシン検出が成功するための条件を明らかにすることが必要である．評価対象方式が QEMU や Raspbian の特徴にどの程度依存しているかは十分に明らかではなく，解明する必要がある．第二に，QEMU 以外のシステムを用いて作られたサンドボックスの検出における評価対象方式の有効性を実験によって評価することが必要である．もし検出精度が高くない場合には，精度を上げるための拡張を検討することになる．第三に，評価対象方式に対する対策の構築が挙げられる．例えば，評価対象方式の処理を意識した実行追跡によって仮想マシン検出の意図を検出したり，プロセスやスレッドに割り当てられる資源の制御によって仮想マシン検出の判定を誤らせたりするなどの対策が考えられる．

謝辞 本研究において，株式会社富士通研究所の小久保博崇氏，筑波大学の忠鉢洋輔氏より有用な助言をいただいた．本研究の一部は JSPS 科研費 17K00179 の助成を受けている．

参考文献

[1] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K. and Zhou, Y.: Understanding the Mirai Botnet, *Proceedings of the 26th USENIX Security Symposium*, pp. 1093–1110 (2017).

[2] Barbosa, G. N. and Branco, R. R.: Prevalent Characteristics in Modern Malware, Black Hat USA 2014 (2014).

[3] Blog, F. S. L.: Locky Returned With A New Anti-VM Trick, `https://blogs.forcepoint.com/security-labs/locky-returned-new-anti-vm-trick` (2016).

[4] Branco, R. R., Barbosa, G. N. and Neto, P. D.: Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies, Black Hat USA 2012 (2012).

[5] Cozzi, E., Graziano, M., Fratantonio, Y. and Balzarotti, D.: Understanding Linux Malware, *Proceedings of the 39th IEEE Symposium on Security and Privacy* (2018).

[6] Deng, Z., Zhang, X. and Xu, D.: SPIDER: Stealthy Binary Program Instrumentation and Debugging Via Hardware Virtualization, *Proceedings of the 29th Annual Computer Security Applications Conference*, pp. 289–298 (2013).

[7] Dinaburg, A., Royal, P., Sharif, M. and Lee, W.: Ether: Malware Analysis via Hardware Virtualization Extensions, *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp. 51–62 (2008).

[8] Dr.Web: Doctor Web examines two Linux Trojans, `https://news.drweb.com/show/?i=11320&lng=en` (2017).

[9] Franklin, J., Luk, M., McCune, J. M., Seshadri, A., Perrig, A. and van Doorn, L.: Remote Detection of Virtual Machine Monitors with Fuzzy Benchmarking, *ACM SIGOPS Operating Systems Review*, Vol. 42, No. 3, pp. 83–92 (2008).

[10] Kawakoya, Y., Iwamura, M. and Itoh, M.: Memory Behavior-Based Automatic Malware Unpacking in Stealth Debugging Environment, *Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software*, pp. 39–46 (2010).

[11] Kedrowitsch, A., Yao, D., Wang, G. and Cameron, K.: A First Look: Using Linux Containers for Deceptive Honeypots, *Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense*, pp. 15–22 (2017).

[12] Kirat, D., Vigna, G. and Kruegel, C.: BareCloud: Bare-metal Analysis-based Evasive Malware Detection, *Proceedings of the 23rd USENIX Security Symposium*, pp. 287–301 (2014).

[13] Kolbitsch, C., Kirda, E. and Kruegel, C.: The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code, *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pp. 285–296 (2011).

[14] Lengyel, T. K., Maresca, S., Payne, B. D., Webster, G. D., Vogl, S. and Kiayias, A.: Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System, *Proceedings of the 30th Annual Computer Security Applications Conference*, pp. 386–395 (2014).

[15] Ning, Z. and Zhang, F.: Ninja: Towards Transparent Tracing and Debugging on ARM, *Proceedings of the 26th USENIX Security Symposium*, pp. 33–49 (2017).

[16] Oyama, Y.: Trends of anti-analysis operations of malwares observed in API call logs, *Journal of Computer Virology and Hacking Techniques*, Vol. 14, No. 1, pp. 69–85 (2018).

[17] Pa, Y. M. P., Suzuki, S., Yoshioka, K., Matsumoto, T., Kasama, T. and Rossow, C.: IoTPOT: Analysing the Rise of IoT Compromises, *Proceedings of the 9th USENIX Workshop on Offensive Technologies* (2015).

[18] Pafish (Paranoid Fish), `https://github.com/a0rtega/pafish/`.

[19] Robin, J. S. and Irvine, C. E.: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, *Proceedings of the 9th USENIX Security Symposium* (2000).

[20] Sophos: Conficker's virtual machine detection, Naked Security, `https://nakedsecurity.sophos.com/2009/03/27/confickers-virtual-machine-detection/` (2009).

[21] Xiao, C., Zheng, C. and Jia, Y.: New IoT/Linux Malware Targets DVRs, Forms Botnet, Palo Alto Networks Blog, `https://researchcenter.paloaltonetworks.com/2017/04/unit42-new-iotlinux-malware-targets-dvrs-forms-botnet/` (2017).

[22] 宮本久仁男，田中英彦：特徴データベースを用いない効率的な仮想マシンモニタ検出方式の提案，情報処理学会論文誌，Vol. 52, No. 9, pp. 2602–2612 (2011).