Computer Security Symposium 2018
22 - 25 October 2018

シンボリック実行を活用したマルウェア解析環境検知機能の回避条件自動抽出の研究

窪 優司[1,†1,a)]　大久保 隆夫[1]

概要：最近のマルウェアは解析環境で実行されていないか確認した上で本来意図した動作を行うことが多いため，マルウェアに組み込まれた解析環境を検知する機能を回避することがマルウェア解析において重要である. 本論文では，解析環境の検知を回避する条件を自動的に抽出するためにシンボリック実行を活用した手法を提案する．提案手法ではシンボリック実行の経路探索を開始するアドレス，到達すべきアドレスおよび回避するアドレスの 3 つのアドレスを指定することで実行経路上の分岐条件を自動的に抽出する．提案手法を 55 種類の解析環境検知機能を実装したプログラムに適用して，解析環境検知機能の回避条件の自動抽出を行って評価する．本提案手法を用いることで，機械的に正確な結果を短時間で抽出できることを示す．

キーワード：シンボリック実行，マルウェア解析，解析環境検知機能

Automatic Extraction of Conditions for Bypassing Malware Anti-Analysis Techniques by Using Symbolic Execution

Yuji Kubo[1,†1,a)]　Takao Okubo[1]

Abstract: It is important in malware analysis to bypass the anti-analysis techniques implemented in malware, since recent malware often performs the intended actions after checking the analysis environment. In this paper, we propose a method that utilizes symbolic execution to automatically extract the conditions for bypassing malware anti-analysis techniques. We propose the method to automatically extract the branch conditions along the execution path by specifying three addresses: the address at which the path exploration starts, the address at which it targets and the address at which it avoids. We evaluate our method by applying it to the program that implements 55 kinds of anti-analysis techniques to obtain the conditions for bypassing them. We demonstrate our method can automatically extract accurate results in a short time.

Keywords: symbolic execution, malware analysis, anti-analysis techniques

1. はじめに

最近のマルウェアは，実装された機能の隠ぺいやウイルス対策ソフトに検知されるまでの時間の長期化などを目的として解析環境を検知して挙動を変化させることが多い．アンチサンドボックス技術，アンチデバッグ技術，コードの暗号化やジャンクコードの挿入などにより研究者や解析者は静的解析および動的解析の双方において対処に時間を要している [1].

そのため，筆者らは Windows 上で動作する実行ファイルを解析対象として，ソフトウェアの試験用データの自動生成などで使用されるシンボリック実行技術を活用し，プログラムが持つ解析環境検知機能を回避するための条件を自動的に抽出する手法の検討を進めている．筆者らは，デバッガ検知機能およびデバッグ作業検知機能を実装したプログラムについて，シンボリック実行技術が備えるパス条

[1] 情報セキュリティ大学院大学
Institute of Information Security
[†1] 現在，警察庁
Presently with National Police Agency of Japan
[a)] mgs168501@iisec.ac.jp

件の抽出と解の計算機能および経路探索機能を活用して検知を回避する条件を短時間かつ自動的に抽出できることを実証した [2]．しかし，解析環境検知機能はデバッガ検知およびデバッグ作業検知以外にも多数存在するため，本稿では既に実証した手法を他のさまざまな解析環境検知機能に対して適用し有効性を評価した．

本稿の貢献は次の 2 点である．

- シンボリック実行技術がプログラムの解析環境検知機能対策に有効であることを実証した．
- シンボリック実行の活用によって従来デバッガなどを使用して解析者が行ってきた解析環境検知機能の回避条件の抽出を機械的かつ短時間で実行できることを実証した．

2. マルウェアの解析環境検知機能

マルウェアが解析環境を検知する手法は多数存在しており，日々新しい手法が考案されマルウェアに実装されているため，すべての解析環境検知手法について述べることはできない．そこで，6.3 における評価対象プログラムである pafish.exe を例に実装された解析環境の検知手法について説明する．pafish.exe は，実際のマルウェアが使用するサンドボックスや解析環境の検知手法を組み込んだデモ用ツールであり，以下に示す複数の観点から解析環境の検知を行う．

- デバッガ検知
  - API の応答
- VM 検知（CPU 情報に基づく）
  - 二地点間の実行時間差
  - レジスタの値
- サンドボックス検知
  - マウスの動作
  - ログオンユーザ名
  - pafish.exe のフルパス名
  - pafish.exe のファイル名
  - ハードディスクのサイズ
  - API の応答
  - プロセッサの数
  - システム起動後の経過時間
  - 仮想ハードディスクドライブからの起動
- フック検知
  - API のアドレス
- Sandboxie [3] 検知
  - ファイルの読み込み
- Wine [4] 検知
  - API のアドレス
  - レジストリ
- VirtualBox [5] 検知
  - レジストリ
  - ファイル
  - ファイルとフォルダ
  - NIC の MAC アドレス
  - デバイス
  - ウィンドウ
  - 共有フォルダ
  - プロセス
- VMware [6] 検知
  - レジストリ
  - ファイル
  - NIC の MAC アドレス
  - ネットワークアダプタの名前
  - デバイス
  - コンピュータのシリアル番号
- Qemu [7] 検知
  - レジストリ
  - プロセッサの名前
- Bochs [8] 検知
  - レジストリ
  - プロセッサの名前
- Cuckoo [9] 検知
  - データ構造のアドレス

3. 関連研究

3.1 シンボリック実行を活用したマルウェアのコードのアンパック

X. Ugarte-Pedrero らは，パックと呼ばれる暗号化処理が施されたマルウェアのコードの復号にシンボリック実行を活用した [10]．実行経路が分岐する際に状態を保存し，各経路について分岐時の状態を再現しながら経路探索を行った．また，特定の API の入出力にテイントと呼ばれるタグを付け，実行に伴うテイントの拡散を通じて復号条件を特定した．他にもマルウェア解析分野特有の複数のヒューリスティック手法を提案し適用した．シンボリック実行技術に最適化処理やヒューリスティック手法を加えることで経路探索機能を改善し，BackPack および Armadillo と呼ばれる実在するパッカーで暗号化されたマルウェアに対して，効率的かつ信頼性のあるコードの復号処理を実証した．

3.2 シンボリック実行を活用した RAT の動作解析

R. Baldoni らは，シンボリック実行フレームワーク angr を基にしたツールを作成し，Remote Access Trojan（リモートアクセス型トロイの木馬：RAT）の指令および指令サーバとの通信プロトコルを自動的に抽出した [11]．外部からの指令は 0x45 で XOR され “@@” で始まる構造を持っていることを解明した．また，マルウェアの内部状態が 2 つのパラメータで規定されており，両者が 1 にセットされるとすべての指令が実行可能になる構造となっていること

を解明した．この結果，マルウェア解析分野におけるシンボリック実行技術の有効性を実証した．

4. シンボリック実行

4.1 シンボリック実行の技術概要

シンボリック実行は，プログラム中の実行可能な経路を自動的に列挙する能力を提供するプログラムの分析技術である．この技術は，1976 年に James C. King により提案された [12]．その後，シンボリック実行に具体値での演算を複合させた K. Sen らによる Concolic Testing [13] や C. Cadar らによる制約解決に改善を加えた Execution Generated Testing（EGT）[14] が考案されて発展した．その後，2011 年に C. Cadar らがバグ発見のための系統立った試験を可能にする近年のシンボリック実行技術を整理し，シンボリック実行が再度注目を集めることとなった [15]．この背景として，シンボリック実行が依存している制約ソルバーの改善やコンピュータの性能の飛躍的な向上などが挙げられる．

具体的な値を使用した実行では，プログラムは特定の入力値に対して実行され，単一の制御フロー経路だけが探索される．そのため，多くの場合で関心のある特徴に関して不十分な解析で終わってしまう．一方，シンボリック実行ではプログラムが異なる入力値に対して取り得る複数の経路を同時に探索することができる [16]．

シンボリック実行では，プログラムは具体的な値ではなく，任意の値を表現するシンボル値と呼ばれる値を入力として受け取る．シンボリック実行エンジンは，各探索経路について経路上の制約を収集して列挙し，経路を記述する式を得る．分岐条件がシンボル値に依存した分岐点に遭遇した場合，現在の式に分岐条件を加えた式と，現在の式に分岐条件を反転した条件を加えた式で記述される二つの状態を生成して実行は分岐する．シンボル値を含む式の評価や，所望の経路に沿ったプログラムの実行に必要な具体的な入力値の生成などに制約ソルバーが使用される．

シンボリック実行エンジンは，プログラム中の分岐点の数が増加すると探索すべき経路の数が指数関数的に増加するパス爆発と呼ばれる問題点を抱えている．そのため，最新のシンボリック実行エンジンでは探索経路のランダムな選択や，コード網羅率を元にした処理などさまざまなパス爆発対策を備えている．

4.2 注目したシンボリック実行の機能

マルウェア解析への適用に当たり，シンボリック実行が備える以下の 2 つの機能に着目した．

- パス条件の抽出と解の計算機能
  実行経路上の各分岐点における分岐条件（パス条件）を抽出して収集し，すべての分岐条件を満たす解を計算する機能
- 経路探索機能
  実行経路上の各分岐点において実行時に選択される経路とは別の経路についても分岐条件を抽出して探索する機能

5. 提案手法

本節では，解析環境検知機能を実装したプログラムに対してシンボリック実行を適用し，解析環境検知機能の回避条件を抽出するための手法について述べる．提案手法の概要を**図 1** に示した．

5.1 シンボリック実行フレームワーク angr の採用

通常，マルウェアのソースコードは入手できないため，バイナリファイル単体でもシンボリック実行を適用できることが必要である．また，プログラム内の二地点間に存在する分岐点で分岐条件を収集し，自動的に実行経路を探索する機能を備えていることが望ましい．以上の要件を満たし，既にぜい弱性検査やファジング分野で研究実績 [17], [18] があるシンボリック実行フレームワーク angr [19] を本稿では採用することとした．

5.2 解析環境検知機能の位置の特定

解析対象のプログラムを逆アセンブルし，解析環境検知機能がプログラムのどの位置に存在するのか確認する．解析環境検知機能はプログラムの持つ機能の隠ぺいを目的としていることが多いため，不正な動作を行う前に解析環境検知機能を実行すると仮定することが自然である．そのため，プログラムの開始直後から `main` 関数の冒頭付近に特に着目すると発見しやすい．

5.3 開始アドレスと到達アドレスの指定

解析環境検知機能の位置を特定できたら，シンボリック実行を開始するアドレスと到達すべきアドレスを指定する．開始アドレスは解析環境検知機能の手前に設定し，到達アドレスは解析環境検知機能の回避後に到達すべきアドレスを設定した．

5.4 回避アドレスの指定

本稿では解析環境検知機能の回避条件の抽出を目的としているため，解析環境検知機能で検知されない場合の実行経路に着目している．したがって，解析環境検知機能で検知される場合の実行経路は，シンボリック実行を行なっても回避条件の抽出に直接寄与せず，むしろ解析に余計な時間を費やすことになる．そこで，解析環境検知機能で検知される場合の実行経路については，その経路上のアドレスを回避アドレスとして指定し，シンボリック実行の探索対象経路から除外した．

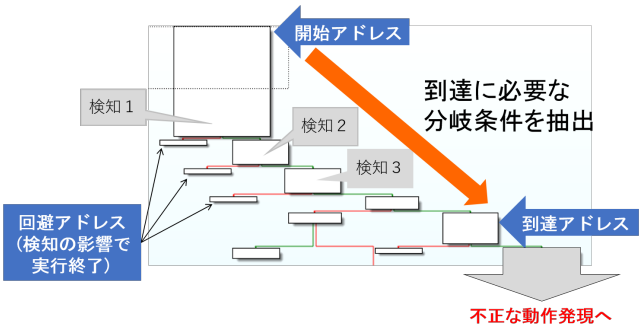

図 1 提案手法の概要説明図

5.5 分岐条件の抽出と解析環境検知機能の回避条件の理解

シンボリック実行により開始アドレスから到達アドレスまでの間に存在する分岐点における分岐条件をすべて収集して列挙する．列挙された条件から解析環境検知機能に関連する条件を抽出し，検知機能の内容を理解する．その上で解析環境検知機能を回避できる環境を準備してプログラムを実行すれば不正な動作が発現することになる．

6. 評価

6.1 評価対象プログラム

評価実験では，マルウェアが使用するサンドボックスや解析環境の検知手法を組み込んだデモ用ツール Pafish (Paranoid Fish) を評価対象のプログラムに選定した．Pafish はオープンソースのプロジェクトとして，実行ファイルの pafish.exe だけでなくそのソースコードも公開されている [20]．pafish.exe は 32 ビットの Windows の実行ファイルである．

本プログラムは，合計 55 種類の解析環境検知手法を実装しており，検知機能の種類と実装された数の内訳を**表 1** に示す．例として，pafish.exe を仮想マシーンの Windows 7 SP1（64 ビット）上で実行したところ**図 2** の結果が得られた．一つの検知項目について解析環境を検知した際には “traced!” と表示し，検知しなかった際には “OK” と表示する．本プログラムは解析環境の検知結果を表示するだけでマルウェアとはいえないが，プログラムに組み込まれた解析環境検知機能はマルウェアが実際に使用する手法を取り入れたものであり，評価実験の対象プログラムとして適切であると考え選定した．

6.2 実験環境

提案手法の評価には，32GB のメモリおよび Intel Core i7 4GHz の CPU を搭載して OS が macOS High Sierra 10.13.4 である iMac Late 2015 をホストマシーンとし、VMware Fusion Pro 8.5.10 を仮想化ソフトとして使用した．仮想マシーンは 4GB のメモリを割り当てた Ubuntu 16.04（64 ビット）を使用し，シンボリック実行フレームワークとして angr 7.8.2.21 を使用した．

表 1 評価対象プログラムの解析環境検知機能一覧

検知機能の種類	実装された数
デバッガ検知	2
VM 検知（CPU 情報に基づく）	4
サンドボックス検知	12
フック検知	2
Sandboxie 検知	1
Wine 検知	2
VirtualBox 検知	17
VMware 検知	8
Qemu 検知	3
Bochs 検知	3
Cuckoo 検知	1

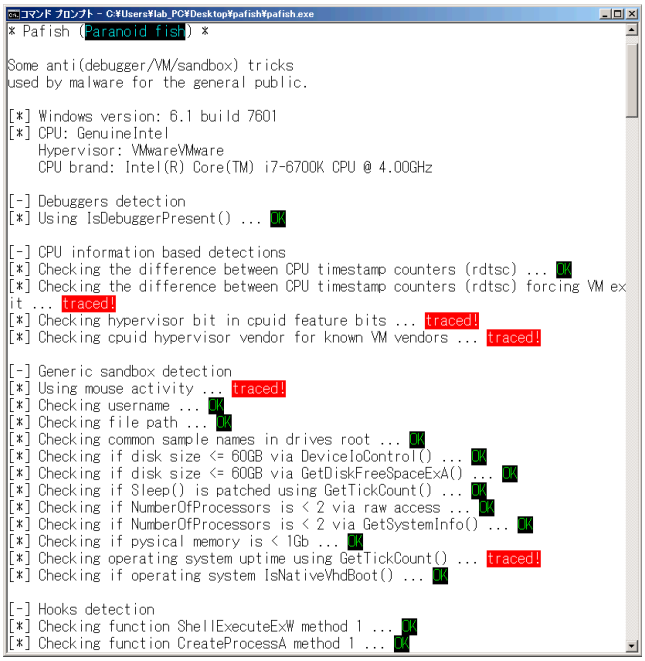

図 2 pafish.exe を Windows 7 SP1（64 ビット）で実行した結果

6.3 実験

本実験では，pafish.exe に実装された合計 55 種類の解析環境検知機能を回避するための条件を抽出した．pafish.exe は，一つの検知項目について解析環境を検知した際には “traced!” と表示し，検知しなかった際には “OK” と表示する．したがって，本実験の目的は pafish.exe が備える各解析環境検知手法に対してシンボリック実行を適用し “OK” と表示させるための条件を求めることである．

公開されているソースコード [20] を確認すると，すべての解析環境検知機能は main 関数内に存在していることが判明した．当初，開始アドレスを main 関数内の最初の解析環境検知機能の直前に，到達アドレスを main 関数内の最後の解析環境検知機能の直後に設定して開始アドレスから到達アドレスに至るために満たすべき分岐条件の抽出を試みたところ，実行環境の動作が著しく遅くなり実行不能な状態に陥った．この原因を調査すると，分岐条件の累積にともない計算量とリソースの使用量が非常に増加したた

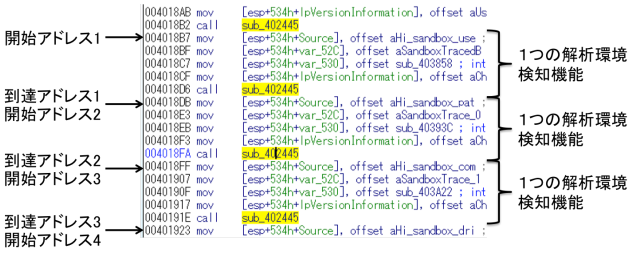

図 3 pafish.exe の main 関数内部のコード

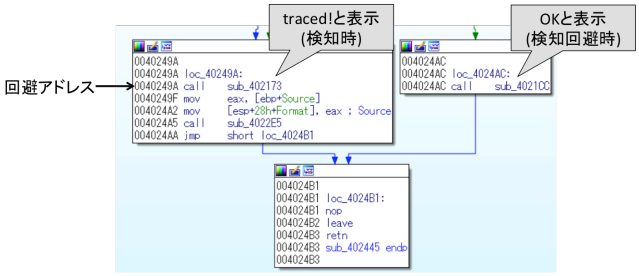

図 4 pafish.exe の検知結果表示関数 sub_402445 内部のコード

めであった．各解析環境検知機能は，互いに相関関係はなく独立した検知手法を直列に実行しているだけである．そこで，一度のシンボリック実行ですべての解析環境検知機能を扱うのではなく，解析環境検知機能ごとに検知の回避条件を抽出し，この作業を pafish.exe に実装された解析環境検知機能の総数である 55 回分繰り返すことにした．

pafish.exe の `main` 関数を逆アセンブルすると，**図 3** のように解析環境検知機能ごとに類似したコード列が直列に並んでいることがわかる．`call sub_402445` という文が繰り返し実行されているが，関数 `sub_402445` は検知成功時に “`traced!`”，検知失敗時に “`OK`” と表示する関数である．関数 `sub_402445` を逆アセンブルすると，関数の末尾付近に**図 4** のようにアドレス 0x40249A で “`traced!`”，アドレス 0x4024AC で “`OK`” と表示する関数を呼び出していて，必ずどちらかの経路を通過することが判明した．

以上のことから，シンボリック実行を行うにあたり，すべての解析環境検知機能に対して共通で回避アドレスを関数 `sub_402445` 内部の “`traced!`” と表示する関数を呼び出している文のアドレス，すなわち 0x40249A を設定することにした．次に，解析環境検知機能ごとに開始アドレスを `main` 関数内の関数 `sub_402445` を呼び出している文の直後のアドレスに，到達アドレスを次の関数 `sub_402445` を呼び出している文の直後のアドレスに設定した．このように設定すると，図 3 のように $1 \leq i \leq 54$ の整数 i に対して，i 番目の解析環境検知機能の到達アドレスが $i+1$ 番目の解析環境検知機能の開始アドレスに一致することになる．以上のアドレス設定でシンボリック実行を行うと，解析環境検知機能ごとに “`traced!`” と表示する関数の実行を回避して検知作業を終える．すなわち，“`OK`” と表示する関数を必ず実行して検知作業を終えることになるため，検知回避条件を抽出できる．

実験の結果を**表 2** に示す．検知回避条件の抽出結果については，抽出できた手法は ✓，抽出できなかった手法は ×，抽出できたが不十分な条件である手法は △ で示している．

7. 実験結果

7.1 実験結果の概要

表 2 の実験結果から，55 種類の解析環境検知手法のうち回避可能な条件を抽出できたのは ✓ と △ の場合を合計した 45 件で全体の約 81.8%を占める．検知の回避は可能だが検知手法の主旨とは異なる条件を抽出した △ の場合を除くと 34 件であり全体の約 61.8%を占める．実行時間については，55 個の解析環境検知機能に対して全体で約 665.6 秒であり，1 個の解析環境検知機能あたり約 12.6 秒であった．

7.2 解析環境検知機能の回避条件の抽出に失敗した場合

本節では，解析環境検知機能の回避条件の抽出に失敗した場合の原因について述べる．

- レジスタの値確認（VM 検知）
  EAX=1 の状態で `cpuid` 命令を実行した結果 ECX の 31 ビット目が 1 であれば仮想環境として検知する．angr では，`cpuid` 命令の動作を模擬する際に ECX=0 とするため，ECX の 31 ビット目は常に 0 となり，検知は自動的に回避されることとなる．そのため，本手法に対する検知条件は抽出できなかった．
- レジスタの値確認（VM 検知）
  EAX=0x40000000 の状態で `cpuid` 命令を実行すると，EBX，ECX および EDX レジスタに CPU の製造会社を示す文字列 12 文字が分割されて格納されるため，当文字列を既知のハイパーバイザーに対する文字列と比較して合致すれば仮想環境として検知する．angr では，`cpuid` 命令の動作を模擬する際に EBX=0，ECX=0，EDX=0x8001BF とするため，ハイパーバイザーを示す文字列が得られることはなく，検知は自動的に回避されることとなる．そのため，本手法に対する検知条件は抽出できなかった．
- 仮想ハードディスクドライブからの起動（サンドボックス検知）
  `GetProcAddress` 関数を使用して `kernel32.dll` から `IsNativeVhdBoot` 関数のアドレスを取得して実行し，その結果から仮想環境を検知する．angr では，取得するアドレスは固定値 0 であり，`IsNativeVhdBoot` 関数は実行できなかった．したがって，本検知手法の回避条件の抽出には至らなかった．
- API のアドレス（フック検知）
  `ShellExecuteExW` 関数または `CreateProcessA` 関数が読み込まれた 4 バイトのアドレスのうち，最上位の

表 2 検知回避条件の抽出結果

解析環境検知機能の種類	条件の抽出結果（件数）			合計
	✓	△	×	
デバッガ検知	2			2
VM 検知（CPU 情報に基づく）	2		2	4
サンドボックス検知	7	4	1	12
フック検知			2	2
Sandboxie 検知			1	1
Wine 検知	1		1	2
VirtualBox 検知	14	3		17
VMware 検知	5	3		8
Qemu 検知	2		1	3
Bochs 検知	1		2	3
Cuckoo 検知		1		1
合計	34	11	10	55

1 バイトが 0x8b かつ次の 1 バイトが 0xff であれば検知回避，それ以外なら検知する．angr では，上述の関数が読み込まれるアドレスの最上位の 1 バイトが固定値 0 であるため必ず検知されることになり，本検知手法の回避条件の抽出には至らなかった．

- ファイルの読み込み（Sandboxie 検知）
  `GetModuleHandle` 関数を使用して `sbiedll.dll` ファイルのハンドルを取得できれば解析環境として検知する．angr では，取得するハンドルは固定値 0 であるため，`sbiedll.dll` ファイルのハンドルを取得できない．したがって，自動的に検知を回避することになるため，回避条件の抽出には至らなかった．
- API のアドレス（Wine 検知）
  `GetModuleHandle` 関数を使用して `kernel32.dll` ファイルのハンドル取得後に `GetProcAddress` 関数を使用して `wine_get_unix_file_name` 関数のアドレスを取得できれば解析環境として検知する．angr では，取得する `kernel32.dll` のハンドルは固定値 0 であるため，自動的に検知を回避することになる．したがって，本検知手法の回避条件の抽出には至らなかった．
- プロセッサの名前（Qemu 検知，Bochs 検知）
  EAX=0x80000002, 0x80000003 および 0x80000004 の状態で `cpuid` 命令を実行すると，CPU のブランドを示す文字列 48 文字が EAX の値ごとに 16 文字で分割される．取得した 16 文字の文字列はさらに EAX，EBX，ECX および EDX レジスタに 4 文字ずつ分割して格納される．取得した CPU のブランドを示す文字列に以下の特定の文字列が含まれると各解析環境として検知する．
  - Qemu 検知
    - “`QEMU Virtual CPU`”
  - Bochs 検知
    - “`AMD Athlon(tm) processor`”
    - “`␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Intel(R)␣Pentium(R)␣4␣CPU␣␣␣␣␣␣␣␣␣`”

  angr では，`cpuid` 命令の動作を模擬する際に EAX=0x543, EBX=0, ECX=0, EDX=0x8001BF とするため，検知対象の CPU のブランドを示す文字列が得られることはなく，検知は自動的に回避されることとなる．そのため，本手法に対する検知条件は抽出できなかった．

8. 考察

8.1 実験結果について

本節では，検知の回避条件の抽出結果について抽出できた場合，抽出できなかった場合と不十分な抽出に至った場合について考察する．

- 回避条件を抽出できた場合（✓ の場合）
  特定のファイルやレジストリの存在を確認する手法について条件を抽出できた場合が多かった．また，特定のデバイスやウィンドウの存在についての条件を抽出できた場合もあった．正しく回避条件を抽出できた場合は，呼び出した関数の返り値が検知対象物の存在の有無を示していることが多いため，条件を反転させて対象物が存在しない場合の返り値に合致する条件を求めれば検知を回避できることになる．
- 回避条件を抽出できなかった場合（× の場合）
  `GetProcAddress` 関数，`GetModuleHandle` 関数や `cpuid` 命令の返り値に応じて解析環境を検知する手法で回避条件を抽出できなかった．angr は，これらの関数または命令を直接実行せず動作を模擬して偽の応答を返すが，その返す応答の内容が不適切なため条件の抽出に失敗した．本来は，関数や命令に渡された引数の内容に応じて返り値は変化するが，angr は常に固定値を返すため自動的に検知を回避して条件の抽出に至らなかった．

シンボリック実行において，関数やアセンブリ命令本体のシンボリック実行を行うと探索すべき経路が増大し，経路探索に失敗して分岐条件の抽出に至らない．そのため angr では，特定の関数やアセンブリ命令の内部のコードはシンボリック実行を行わずに偽の応答を返すことにしている．この仕組みによって複雑な関数や命令内部を調査せずに迅速なシンボリック実行が可能になる．一方で，適切でない応答が分岐条件の抽出に失敗する原因となる場合があることが判明した．

- 不十分な回避条件の抽出に至った場合（△ の場合）
  プロセス一覧やデバイス一覧を取得した後で特定のプロセスやデバイスを検索して検知する手法に対しては，そもそもプロセス一覧やデバイス一覧を取得できないことを回避条件として出力して終了したため，特定のプロセスやデバイスとの比較に至らなかった．同様に，ハードディスクの容量が 60GB を下回ると解析環境として検知する手法に対しても，調査対象のハードディスクが存在しないことを回避条件として出力し，容量の比較に至らなかった．以上の結果は，`Pafish` のソースコードと比較すると，確かに上述の条件が満たされれば検知が回避される記述になっているため，シンボリック実行の結果としては適切な条件を抽出している．不十分な条件の抽出に至った理由は，一旦検知を回避する条件を抽出した時点で以降に存在する回避条件を探索せず終了してしまうためである．検知を回避する条件を発見しても別の条件を発見するまで探索を継続すればよいが，探索すべき経路数の増大から探索不能に至る可能性もあるため，適切な条件の抽出は必ずしも容易ではない．

単純に検知を回避することだけを目的とした場合は，8 割以上の検知手法に対して回避条件を抽出でき，おおむね良好な結果が得られた．しかし，検知手法の主旨に沿った適切な回避条件を抽出できた場合は約 6 割にとどまったため，改善の余地が多分にある．

実行時間については，全体の実行時間が約 665.6 秒であり現実的な時間で実行を完了できたといえる．ただし，実際のマルウェアではシンボリック実行技術が処理に時間を要するループ構造の存在や多数の分岐点の存在によって探索すべき経路が増大して現実的な時間で処理を完了できない場合も想定されるため，これらの時間を要する演算の処理に工夫が必要となる．

8.2 課題

本稿で適用した angr による解析環境検知機能の回避条件の抽出における課題を述べる．

- 開始アドレス，到達アドレスおよび回避アドレスの決定手法
  本稿では，開始アドレス，到達アドレスおよび回避アドレスはプログラムを逆アセンブルして指定したが，プログラムが変われば別途解析し直す必要があるため，これらのアドレスを自動的に抽出できる手法を考案する必要がある．
- 他の解析環境検知機能への対応
  マルウェアが使用する解析環境検知機能には，本稿で扱った手法以外にもさまざまな手法が存在する．angr によるシンボリック実行でどの程度対処できるか確認するとともに，条件の抽出に失敗した場合に原因を探りどのような対処が可能か調査する必要がある．
- コードの暗号化への対応
  パックと呼ばれる実行コードの暗号化が施されている場合，適切な開始アドレスや到達アドレスはマルウェアを実行している途中でないと指定できない．現在 angr は暗号化されたコードの処理ができないため，別途パック対策用ソフトウェアの導入や angr の機能の増強などの対策を検討する必要がある．
- シンボリック実行技術に対する妨害への対応
  マルウェアの作者がシンボリック実行技術を妨害するために，無限ループや条件分岐を必要以上に実装して探索すべき経路を増加させることが考えられる．この場合，不要なループを処理しない工夫や深度優先または幅優先などの探索方針の決定が必要である．

9. おわりに

本稿では多数の解析環境検知機能を持つプログラムに対して，シンボリック実行フレームワーク angr を活用して解析環境検知機能を回避するための条件の自動的な抽出を行い，おおむね良好な結果が得られた．条件の抽出に失敗した場合は，angr の関数や命令の模擬の失敗が理由であり，シンボリック実行技術自体はマルウェア解析に有効な手法であることを示すことができた．

従来は，解析者によるデバッガや逆アセンブラなどを使用した慎重かつ正確な作業が必要で時間も要していたが，angr やコンピュータの演算能力を活用し現実的な時間で処理することができた．また，人間が手動で行うことによる誤解や誤操作を排除して機械的に正確な結果を得ることができた．

シンボリック実行を活用した解析環境検知機能の回避において最も重要な点は，解析環境の有無を判断する分岐条件である．解析環境検知の内容がたとえ未知の手法であったとしても分岐条件さえ取得できれば，その条件を満たすようにメモリやレジスタの値を調整することで自動的に回避することができる可能性がある．8.2 で述べたように解決すべき課題は多いが，本稿の提案手法は人間が一切手を加えることなく解析環境検知機能を回避してマルウェアの機能を明らかにできる完全な自動解析システムの実現に向けて有用な技術であると考えられる．

今後は，現在バイナリファイルごとに設定している開始アドレスおよび到達アドレスの指定を自動化するとともに，多様な解析環境検知機能を扱うことができるよう，より汎用性のある手法へ拡張することを目標に研究を進めていきたい．

参考文献

[1] ICS-CERT: Malware Trends, National Cybersecurity and Communications Integration Center (NCCIC), Department of Homeland Security (DHS) (online), available from ⟨https://ics-cert.us-cert.gov/sites/default/files/documents/NCCIC_ICS-CERT_AAL_Malware_Trends_Paper_S508C.pdf⟩ (accessed 2018-05-31).

[2] 窪　優司，大久保隆夫：シンボリック実行を活用したマルウェア解析作業の効率化の研究，技術報告 39，情報セキュリティ大学院大学 (2018).

[3] Sandboxie Holdings, LLC: Sandboxie - Sandbox software for application isolation and secure Web browsing, Sandboxie Holdings, LLC (online), available from ⟨https://www.sandboxie.com/⟩ (accessed 2018-06-26).

[4] Wine Project: WineHQ - Run Windows applications on Linux, BSD, Solaris and macOS, Wine Project (online), available from ⟨https://www.winehq.org/⟩ (accessed 2018-06-26).

[5] Oracle: Oracle VM VirtualBox, Oracle (online), available from ⟨https://www.virtualbox.org/⟩ (accessed 2018-06-26).

[6] VMware, Inc: VMware によるデスクトップ、サーバ、アプリケーション、パブリッククラウドおよびハイブリッドクラウドの仮想化，VMware, Inc（オンライン），入手先 ⟨https://www.vmware.com/jp.html⟩（参照 2018-06-26）.

[7] Bellard, F.: QEMU, Fabrice Bellard (online), available from ⟨https://www.qemu.org/⟩ (accessed 2018-06-26).

[8] The Bochs Project: bochs: The Open Source IA-32 Emulation Project (Home Page), The Bochs Project (online), available from ⟨http://bochs.sourceforge.net/⟩ (accessed 2018-06-26).

[9] Cuckoo Sandbox: Cuckoo Sandbox - Automated Malware Analysis, Cuckoo Sandbox (online), available from ⟨https://cuckoosandbox.org/⟩ (accessed 2018-06-26).

[10] Ugarte-Pedrero, X., Balzarotti, D., Santos, I. and Bringas, P. G.: RAMBO: Run-Time Packer Analysis with Multiple Branch Observation, *Detection of Intrusions and Malware, and Vulnerability Assessment* (Caballero, J., Zurutuza, U. and Rodríguez, R. J., eds.), Cham, Springer International Publishing, pp. 186–206 (2016).

[11] Baldoni, R., Coppa, E., D'Elia, D. C. and Demetrescu, C.: Assisting Malware Analysis with Symbolic Execution: A Case Study, *Cyber Security Cryptography and Machine Learning* (Dolev, S. and Lodha, S., eds.), Cham, Springer International Publishing, pp. 171–188 (2017).

[12] King, J. C.: Symbolic Execution and Program Testing, *Commun. ACM*, Vol. 19, No. 7, pp. 385–394 (online), DOI: 10.1145/360248.360252 (1976).

[13] Sen, K.: Concolic Testing, *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, New York, NY, USA, ACM, pp. 571–572 (online), DOI: 10.1145/1321631.1321746 (2007).

[14] Cadar, C. and Engler, D.: Execution Generated Test Cases: How to Make Systems Code Crash Itself, *Model Checking Software* (Godefroid, P., ed.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 2–23 (2005).

[15] Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N. and Visser, W.: Symbolic Execution for Software Testing in Practice: Preliminary Assessment, *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, New York, NY, USA, ACM, pp. 1066–1071 (online), DOI: 10.1145/1985793.1985995 (2011).

[16] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C. and Finocchi, I.: A Survey of Symbolic Execution Techniques, *ACM Comput. Surv.*, Vol. 51, No. 3 (2018).

[17] Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C. and Vigna, G.: Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware, *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015* (2015).

[18] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C. and Vigna, G.: Driller: Augmenting Fuzzing Through Selective Symbolic Execution, *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016* (2016).

[19] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Krügel, C. and Vigna, G.: SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis, *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pp. 138–157 (2016).

[20] Ortega, A.: pafish: Pafish is a demonstration tool that employs several techniques to detect sandboxes and analysis environments in the same way as malware families do., Alberto Ortega (online), available from ⟨https://github.com/a0rtega/pafish⟩ (accessed 2018-07-12).