Vol.2018-CSEC-82 No.39
Vol.2018-SPT-29 No.39
2018/7/26

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

窪 優司[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妨害機能を理解して回避することがマルウェア解析において重要である．シンボリック実行を活用することで，解析を妨げる制約を列挙して回避のための条件が明らかになる可能性がある．本研究ではシンボリック実行を活用し，マルウェアの解析妨害機能を回避する手法を提案し評価を行った．

キーワード：シンボリック実行, RAT, マルウェア解析, 解析妨害技術

Improvement of Malware Analysis By Using Symbolic Execution

Yuji Kubo[1,a)]　Takao Okubo[1]

Abstract: Understanding and circumventing anti-analysis techniques embedded in malware binaries are crucial for malware analysis because recent malware often reveals the intended actions after checking analysis environment. With symbolic execution, the constraints for hindering the analysis could be listed and revealed. The goal of this research is circumventing the anti-analysis techniques with symbolic execution.

Keywords: Symbolic execution, RAT, malware analysis, anti-analysis techniques

1. はじめに

最近のマルウェアは，自身の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等を目的として解析妨害機能を備えていることが多い．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等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]．

本稿では，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等に活用されているシンボリック実行技術が備えるパス条件の抽出と解の計算機能および経路探索機能を活用し，プログラムが持つ解析妨害機能を回避するための条件を抽出することを目的として手法の提案と評価を行った．

[1] 情報セキュリティ大学院大学
Institute of Information Security
[a)] mgs168501@iisec.ac.jp

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妨害機能対策に適用できることを実証した．
- シンボリック実行を活用すれば，従来デバッガ等を使用して解析者が行う解析妨害機能の回避条件の抽出を短時間で実行できることを実証した．

2. マルウェアの解析妨害機能

2.1 マルウェアの解析妨害機能概要

現在，アンチサンドボックス技術，アンチデバッグ技術および耐解析技術の増加により，マルウェアの静的解析と動的解析が妨げられ，研究者や解析者の対処に遅延が生じさせられている [1]．一般に，マルウェアは以下のような解析妨害技術で保護されている．

- 解析環境検知
  - 解析ツールの動作・存在確認

– 仮想環境の動作確認
– インターネットへの接続の確認
- コード難読化
– コード圧縮
– コード暗号化

本稿では，前者の解析環境検知による解析妨害機能の中でも，デバッガまたはデバッグ作業を検知する手法を対象にした．この理由は，解析者がマルウェアの挙動を理解するためにデバッガを使用していることをマルウェアの作者は熟知しており，できるだけ解析作業を遅らせる目的でデバッガまたはデバッグ作業を検知する妨害を組み込むことが多いためである．

2.2 デバッガ検知機能

プログラムがデバッガを検知する方法として，デバッガの使用の有無により戻り値が異なる IsDebuggerPresent 関数等の Windows の API を使用する方法がある．しかし，API 呼出しをフックして偽の応答を返すことで検知を回避できるため [2]，API を使用せずに手動でプロセス内のメモリを確認し，デバッガの存在を検知する手法が知られている．

各実行中のプロセスに対して，プロセスに関連する情報が記録された Process Environment Block（PEB）と呼ばれるデータ構造が Windows によって管理されており，プロセスがデバッグされているか示す情報も含まれている．PEB のデータ構造は Microsoft 社により公開されているが，一部のデータは OS が内部で使用するデータとして詳細が明らかにされていない [3]．プロセスの実行中に PEB の先頭アドレスは FS レジスタの先頭から 0x30 バイト進んだ位置に記録されている．

本節では，PEB 内のデータを使用した 3 種類のデバッガ検知手法について述べる．

2.2.1 BeingDebugged フラグの確認

BeingDebugged フラグは，PEB の先頭から 2 バイト進んだ位置に存在するデータであって，プロセスがデバッグされていると 1，デバッグされていないと 0 が記録される．

2.2.2 ProcessHeap 内フラグデータの確認

Microsoft 社が公開している PEB のデータ構造の情報には明示されていないが，ProcessHeap として知られているデータ構造が PEB 内に存在する．ProcessHeap には，プロセスの実行時に最初に割り当てられたヒープ領域がデバッガ内で作成されたかどうかを示す ForceFlags フラグと呼ばれるデータがあり，デバッガ上で実行されていると 0 でない値になる．

ProcessHeap のメモリアドレスは，PEB の先頭から 0x18 バイト進んだ位置に記録されている．ForceFlags フラグデータは，32 ビットアプリケーションにおいて Windows XP では ProcessHeap の先頭から 0x10 バイト進んだ位置に記録されている [2]．

2.2.3 NTGlobalFlag フラグの確認

デバッガ上でプロセスを実行すると，デバッガを使用しない場合とはメモリ内のヒープ領域の作成方法が異なる．Microsoft 社が公開している PEB のデータ構造の情報には明示されていないものの，システムがどのようにヒープ領域を作成するか決定するために使用する NTGlobalFlag フラグと呼ばれるデータが PEB の先頭から 0x68 バイト進んだ位置に記録されている．このデータの値が 0x70 であればデバッガ上で実行されていることになる [2]．

2.3 デバッグ作業検知機能

一般にデバッグ作業を行うと，人間によるデータの確認等の影響によりプロセスの実行速度は通常の実行に比べて遅くなる．この特徴に着目した時間差確認によるデバッグ作業の検知手法が存在する．すなわち，適当なコード位置でタイムスタンプを取得し，何らかの作業を行なった後に再びタイムスタンプを取得して，二つのタイムスタンプの差分を計算する．この差分が通常実行した場合に比べて大きいと判断できた場合，デバッグ作業をしていると推定できるという手法である [2]．本節では，時間差確認による 3 種類のデバッグ作業の検知手法について述べる．

2.3.1 QueryPerformanceCounter 関数を使用した時間差確認

`QueryPerformanceCounter` 関数は，プロセッサ内の高分解能パフォーマンスカウンタの現在の値を取得する関数である [4]．この関数を二度実行して差分を取得し，通常実行した場合と比べて長い時間が経過している場合はデバッグ作業を行っているものとして検知する．

2.3.2 GetTickCount 関数を使用した時間差確認

`GetTickCount` 関数は，システムを起動した後の経過時間をミリ秒（ms）単位で取得する関数である [5]．この関数を二度実行して差分を取得し，通常実行した場合と比べて長い時間が経過している場合はデバッグ作業を行っているものとして検知する．

2.3.3 rdtsc 命令を使用した時間差確認

`rdtsc` 命令（オペコードは `0x0F31`）は，プロセッサのタイムスタンプ・カウンタの現在の値を `EDX:EAX` レジスタにロードする命令である [6]．この命令を二度実行して差分を取得し，通常実行した場合と比べて長い時間が経過している場合はデバッグ作業を行なっているものとして検知する．

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 [16]．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 [7]，2011 年に C. Cadar らにより再度注目を集めることとなった [8]．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等が挙げられる．シンボリック実行は，高いコード網羅率を実現する試験用データの自動生成を目的としてソフトウェアの試験分野で広く用いられている．

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，実行は異なる式で記述される二つの状態を生成して分岐する．この時の二つの式は，現在の式に分岐条件を加えたものと，現在の式に分岐条件を反転した条件を加えたものとして得られる．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等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してしまう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等さまざまなパス爆発対策を備えている．

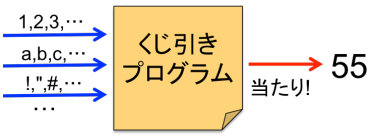

図 1　総当たりによる試験

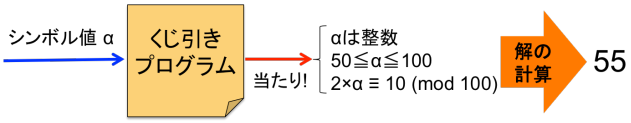

図 2　シンボリック実行による試験

4.2 シンボリック実行の適用例

くじ引きを行うプログラムに対して，当たりになる入力値を求める試験を考える．このプログラムは，一個の入力値に対して当たりが決まる．試験方法は，プログラム内部のコードを確認せずに行うブラックボックス試験とする．

図 1 のように単純な総当たりで求めると，55 を入力した際に当たりとなることがわかる．しかし，55 が当たりであるという事実だけで，他にも当たりとなる値があるかは入力値が取り得るすべての値を入力しない限りわからない．

一方，シンボリック実行を活用した試験では，入力として具体的な値ではなくシンボル値を使用する．すなわち，図 2 のようにシンボル値 α を入力値とすると，当たりとなる時に入力値が満たすべき条件が収集されて列挙される．その後，列挙された条件をすべて満たす値を制約ソルバーにより計算すると，入力値が 55 の時に当たりとなり，かつ 55 以外に当たりとなる値はないことがわかる．

4.3 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を見据えて，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
  パス条件は，入力されたシンボル値に依存したすべての分岐条件を記録したものである．制約ソルバーでパス条件に含まれるすべての条件を満たす解を計算し，解が求まれば特定経路の実行やバグの再現などに活用できる．
- 経路探索機能
  実行経路が分岐する際に，現在の状態で実行される経路とは別の経路についても分岐条件を反転させて探索する機能がある．

5. 提案手法

本節では，解析妨害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，解析妨害機能の回避条件を抽出する手法について述べる．

5.1 シンボリック実行フレームワーク angr の採用

実行ファイルに対しては，ソースコードがないバイナリファイル単体でも扱え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 [12], [13] があるシンボリック実行フレームワーク angr [14] を本研究では採用することとした．

5.2 解析妨害機能の位置の特定

解析対象のプログラムを逆アセンブルし，解析妨害機能がプログラムのどの位置に存在するのか確認する．解析妨害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妨害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

解析妨害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妨害機能の手前に設定し，到達アドレスはすべての解析妨害機能が回避された後に到達すべきアドレスを設定することが望ましい．

5.4 回避アドレスの指定

解析妨害機能で検知されて実行終了へ至るような正常な実行経路以外の経路までシンボリック実行を行なっては解析に余計な時間を費やすことになる．そこで，プログラムを逆アセンブルして明らかに正常な実行経路から外れていると判断できる経路については，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．

なお，回避アドレスの指定による実行時間への影響について考察するため，回避アドレスを指定した場合と指定しない場合の両方について実験を行なった．

5.5 分岐条件の抽出と解析妨害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妨害機能に関連していると思われる条件を抽出し，妨害機能の内容を理解する．その上で解析妨害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．

```
1 SignExt(24L, mem_c0081002_6562_8) == 0x0
2 mem_90_9_32 == 0x0
3 (0xffffff90 + mem_f0021068_10_32) != 0x0
```

図 3 Lab16-01.exe の解析妨害機能回避条件

6. 評価

6.1 実験対象プログラム

評価実験では，書籍 [2] の演習問題 [15] のうち，「Chapter 16: Anti-Debugging」の演習問題である Lab16-01.exe および Lab16-03.exe を解析対象のプログラムに選定した．これらのプログラムは 32 ビットの Windows XP で動作させることを想定している．

こららのプログラムは，厳密にはマルウェアではないかもしれないが，プログラムに組み込まれた解析妨害機能や妨害回避後に発現する RAT としての挙動は実際のマルウェアを模擬したものであり，評価実験の対象プログラムとして問題はないと考え選定した．

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．

6.3 実験 1

本節では Lab16-01.exe に対して angr を使用して解析妨害機能を回避するための条件を抽出した．Lab16-01.exe は，2.2 で述べた 3 種類のデバッガ検知手法を実装しており，いずれかの手法によってデバッガを検知すると，プロセスを終了して自分自身のファイルを削除する．

プログラムを逆アセンブルすると，3 種類のデバッガ検知機能はすべて `main` 関数の冒頭に存在していることが判明した．そこで，シンボリック実行を行うにあたり，開始アドレスを `main` 関数の先頭アドレスに，到達アドレスを 3 つの妨害回避後に到達するアドレスに設定した．この状態で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を抽出したところ**図 3** の結果が得られた．

いずれの条件についても，“`mem_`” で始まる文字列はメモリ内のデータを指すシンボル値であることを表している．例として，1 行目の条件に含まれるシンボル値文字列 `mem_c0081002_6_8` を構成する各要素は以下の情報を示している．

- `mem` メモリ内のデータであることを示す

- c0081002 メモリアドレス 0xc0081002 の値を示す
- 6 angr 内部で使用する識別子を示す
- 8 シンボル値のビット長を示す

なお，この例におけるメモリアドレス 0xc0081002 は，angr がシンボリック実行の過程で便宜的に付与したアドレスであるため，このアドレスに実際に操作を加えても解析妨害機能を回避できない．

しかし，angr がシンボリック実行の過程で出力する履歴を確認することにより，シンボル値 mem_c0081002_6_8 が生成された過程を理解でき，ひいては解析妨害機能を回避する条件を理解できる．付録 A.1 に angr の履歴の関連箇所を抜粋した．

6.3.1 BeingDebugged フラグの確認

図 3 の 1 行目の条件は，angr の履歴 A.1.1 から，PEB のアドレスを取得し,PEB の先頭から 2 バイト進んだ位置に記録された 1 バイト長の BeingDebugged フラグを取得していることがわかる．条件式にある SingExt(24L は，8 ビットのフラグデータに 24 ビットを付加して 4 バイトデータへ符号拡張することを意味している．したがって，1 行目の条件式は，BeingDebugged フラグのデータが 0 であることを要求している．

6.3.2 ProcessHeap 内フラグデータの確認

図 3 の 2 行目の条件は，angr の履歴 A.1.2 から，PEB のアドレスを取得し,PEB の先頭から 0x18 バイト進んだ位置に記録された ProcessHeap のアドレスを取得している．その後 ProcessHeap の先頭から 0x10 バイト進んだ位置に記録された 4 バイト長の ForceFlags フラグを取得していることがわかる．したがって，2 行目の条件式は，ProcessHeap 内の ForceFlags フラグのデータが 0 であることを要求している．

6.3.3 NTGlobalFlag フラグの確認

図 3 の 3 行目の条件は，angr の履歴 A.1.3 から，PEB のアドレスを取得し,PEB の先頭から 0x68 バイト進んだ位置に記録された 4 バイト長の NTGlobalFlag フラグを取得していることがわかる．3 行目の条件式中の値 0xffffff90 を符号なし整数の値から符号付き整数の値に変換すると-0x70 となる．したがって，3 行目の条件式は次のように書き換えることができる．

```
mem_f0021068_10_32 != 0x70
```

すなわち，NTGlobalFlag フラグのデータが 0x70 でないことを要求している．

6.3.4 実験 1 における解析妨害機能回避条件

以上の実験結果から，実験 1 における解析妨害機能を回避するための条件は次の 3 条件であることが判明した．

- BeingDebugged フラグのデータが 0 であること
- ProcessHeap 内の ForceFlags フラグのデータが 0 であること
- NTGlobalFlag フラグのデータが 0x70 でないこと

6.4 実験 2

本節では，Lab16-03.exe に対して angr を使用して解析妨害機能を回避するための条件を抽出した．Lab16-03.exe は，2.3 で述べた 3 種類のデバッグ作業検知手法を実装しており，デバッグ作業を検知した手法によって検知後の挙動が以下のように変化する．

- QueryPerformanceCounter 関数による検知時
  プログラムが正常に動作する際に満たすべきファイル名とは異なる名前に書き換えて正常な動作を妨げる
- GetTickCount 関数による検知時
  意図的に例外を発生させてプログラムを異常終了させる
- rdtsc 命令による検知時
  プログラムを終了し，自分自身のファイルの削除を試みる

プログラムを逆アセンブルすると，3 種類のデバッガ検知機能はすべて main 関数の冒頭に存在していることが判明した．そこで，シンボリック実行を行うにあたり，開始アドレスを main 関数の先頭アドレスに，到達アドレスを 3 つの妨害回避後に到達するアドレスに設定した．この状態で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を抽出したところ**図 4** の結果が得られた．

6.4.1 QueryPerformanceCounter 関数を使用した時間差確認

図 4 の 1 行目の条件は，QueryPerformanceCounter 関数は取得するカウンタの値が 64 ビット長の値であり [4]，条件式中では [31:0] と付加されていることから，一度目と二度目に取得したカウンタの下位 32 ビットを抽出していることがわかる．条件式中の値 0xffffffff を符号なし整数の値から符号付き整数の値に変換すると −1 となる．したがって，条件式は次のように書き換えることができる．

二度目のカウンタ + (−1 × 一度目のカウンタ) ≤ 0x4b0

すなわち，二度取得したカウンタの差分が 0x4b0 以下であることを要求している．

6.4.2 GetTickCount 関数を使用した時間差確認

図 4 の 10 行目の条件は，GetTickCount 関数でシステムを起動した後の経過時間をミリ秒（ms）単位で二度取得し，その差分を求めている．つまり，条件式は次のことを意味している．

二度目の経過時間 − 一度目の経過時間 ≤ 1

すなわち，二度取得した経過時間の差分が 1 ミリ秒以下であることを要求している．

6.4.3 rdtsc 命令を使用した時間差確認

図 4 の 11 行目の条件は，rdtsc 命令は取得するカウンタの値が 64 ビット長の値であり [6]，条件式中では [31:0] と付加されていることから，一度目と二度目に取得したカウンタの下位 32 ビットを抽出していることがわかる．条件式中の値 0xffffffff を符号なし整数の値から符号付き整数

```
1  (QueryPerformanceCounter_result_6_64[31:0] + (0xffffffff * QueryPerformanceCounter_result_4_64[31:0]))
       <= 0x4b0
2  mem_1_11_8 == 112
3  mem_2_13_8 == 101
4  mem_3_15_8 == 111
5  mem_4_17_8 == 46
6  mem_5_19_8 == 101
7  mem_6_21_8 == 120
8  mem_7_25_8 == 101
9  mem_8_31_8 == 0
10 (GetTickCount_result_71_32 - GetTickCount_result_68_32) <= 0x1
11 (RDTSC_75_64[31:0] + (0xffffffff * RDTSC_73_64[31:0])) <= 0x7a120
```

図 4　Lab16-03.exe の解析妨害機能回避条件

の値に変換すると −1 となる．したがって，条件式は次のように書き換えることができる．

二度目のカウンタ + (−1 × 一度目のカウンタ) ≤ 0x7a120

すなわち，二度取得したカウンタの差分が 0x7a120 以下であることを要求している．

6.4.4　特定のファイル名であることを要求する妨害

図 4 の 2 行目から 9 行目までの条件は，シンボル値の名前が “mem” で始まり，末尾の値が 8 であることから，それぞれの式はメモリ中の 1 バイトの値が満たすべき条件を示していることがわかる．さらに，“mem_” 直後のメモリアドレスに相当する値が 1 から順にインクリメントされていることから，2 行目の条件式が先頭の文字を表し，9 行目の条件式が末尾の文字を表していることがわかる．

各条件式の先頭の文字から右辺の値を抽出すると，[112，101，111，46，101，120，101，0] となる．この数値を ASCII コードとして文字に変換すると，[p，e，o，.，e，x，e，\x00] となる．シンボル値の生成とファイル名の各文字が満たすべき条件について，付録 A.2 に angr の履歴の関連箇所を抜粋した．

以上の結果から，既に述べた 3 種類のデバッグ作業検知による妨害に加えて，プログラムのファイル名が “peo.exe” であることを要求している．

6.4.5　実験 2 における解析妨害機能回避条件

以上の実験結果から，実験 2 における解析妨害機能を回避するための条件は次の 4 条件であることが判明した．

- QueryPerformanceCounter 関数で取得したカウンタの差分が 0x4b0（10 進数で 1,200）以下であること
- GetTickCount 関数で取得した経過時間の差分が 1 以下であること
- rdtsc 命令で取得したカウンタの差分が 0x7a120（10 進数で 500,000）以下であること
- ファイル名が “peo.exe” であること

表 1　実験結果

	実験 1		実験 2	
解析妨害機能の数	3		4	
抽出した条件の数	3		11	
回避アドレスの指定	有	無	有	無
実行時間 （秒）	0.407	0.481	33.197	41.915

7.　考察

7.1　実験結果について

実験 1 と実験 2 の結果をまとめると**表 1** のとおりとなった．いずれの実験も組み込まれた 3 種類のデバッガまたはデバッグ作業検知による解析妨害機能を回避する条件を抽出できた．実験 2 については，正常に動作するために設定すべきファイル名についても抽出できた．

実行時間について，実験 2 が実験 1 より時間を要した理由は，ファイル名に関する条件を抽出する際のループ構造の処理に時間を要したためと考えられる．また，回避アドレスを指定すると，指定しない場合に比べて実行時間が短縮された．これは 5.4 で述べたように回避アドレスを指定することで，シンボリック実行で探索すべき経路が限定されて処理を高速に行うことができたためであると考えられる．

シンボリック実行を行う上で主要な設定は開始アドレスと到達アドレスであり，両アドレス間の経路探索や条件抽出は angr が自動的に抽出した．

従来は，解析者によるデバッガや逆アセンブラ等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短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間違いを排除して機械的に正確な結果を得ることができた．

7.2　課題

本稿で適用した angr による解析妨害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．
- 他の解析妨害機能への対応
  マルウェアが使用する解析妨害機能には，本稿で扱ったデバッガまたはデバッグ作業検知機能以外にも仮想環境や解析ツールの検知などさまざまな手法が存在する．また，デバッガまたはデバッグ作業検知機能についても本稿で扱った手法以外の手法も存在する．angr によるシンボリック実行でどの程度対処できるかさらなる調査が必要である．
- コードの暗号化への対応
  パックと呼ばれるコードの暗号化が施されている場合，開始アドレスや到達アドレスはマルウェアの実行中に現れる．現在 angr は暗号化されたコードの処理ができないため，さらなる改善や試験を行い対処する必要がある．
- 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探索方針の決定等の対策が必要である．

8. おわりに

本稿ではデバッガまたはデバッグ作業を検知することによる解析妨害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 を活用して解析妨害機能を回避するための条件を自動的に抽出することができた．従来は，解析妨害機能の内容の理解と妨害回避手法の発見に解析者がデバッガや逆アセンブラ等を使用した慎重かつ正確な作業を行って時間を要していた．本提案手法によって，シンボリック実行のパス条件の抽出機能および経路探索機能を活用することで，解析妨害機能の回避条件を人間が介在することなく機械的かつ短時間に抽出できた．

今後は，現在バイナリファイルごとに設定している開始及び到達アドレスの取得を自動化するとともに，多様な解析妨害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 (online), available from ⟨https://ics-cert.us-cert.gov/sites/default/files/documents/NCCIC_ICS-CERT_AAL_Malware_Trends_Paper_S508C.pdf⟩ (2018.05.31).
[2] Sikorski, M. and Honig, A.: *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, No Starch Press, 1st edition (2012).
[3] Microsoft: PEB structure (online), available from ⟨https://msdn.microsoft.com/en-us/library/windows/desktop/aa813706(v=vs.85).aspx⟩ (2018.06.03).
[4] Microsoft: QueryPerformanceCounter function (online), available from ⟨https://msdn.microsoft.com/ja-jp/library/windows/desktop/ms644904(v=vs.85).aspx⟩ (2018.06.04).
[5] Microsoft: GetTickCount function (online), available from ⟨https://msdn.microsoft.com/ja-jp/library/windows/desktop/ms724408(v=vs.85).aspx⟩ (2018.06.04).
[6] Intel Corporation: IA-32 インテル® アーキテクチャソフトウェア・デベロッパーズ・マニュアル：中巻 B:命令セット・リファレンス N-Z (2004).
[7] James, C. K.: Symbolic Execution and Program Testing, *Commun. ACM*, Vol. 19, No. 7, pp. 385–394 (1976).
[8] 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, pp. 1066–1071 (2011).
[9] Papp, D., Buttyán, L. and Ma, Z.: Towards Semi-automated Detection of Trigger-based Behavior for Software Security Assurance, *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES '17)*, New York, NY, USA, pp. 64:1–64:6 (2017).
[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* Springer International Publishing, pp. 186–206 (2016).
[11] Sen, K.: Concolic Testing, *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, New York, NY, USA, pp. 571–572 (2007).
[12] 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, (2015).
[13] 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, (2016).
[14] 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, pp. 138–157, (2016).
[15] Sikorski, M.: PracticalMalwareAnalysis-Labs (online), available from ⟨https://github.com/mikesiko/PracticalMalwareAnalysis-Labs⟩ (2018.06.04).

[16] 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* Springer International Publishing, pp. 171–188 (2017).

付　　録

A.1 実験 1 における angr の履歴

A.1.1 BeingDebugged フラグの確認

- FS レジスタからの読み込み

```
Reg Read <BV16 0xc008> from fs register
with length(bytes) 2
```

- PEB アドレスの取得

```
Mem Read <BV32 0xc0081000> from
<BV32 0xc0080030> with length(bytes) 4
```

- BeingDebugged フラグの取得

PEB+0x02 のアドレスを計算している

```
Mem Read <BV8
mem_c0081002_6_8{UNINITIALIZED}> from
<BV32 0xc0081002> with length(bytes) 1
```

A.1.2 ProcessHeap 内フラグデータの確認

- FS レジスタからの読み込み

```
Reg Read <BV16 0xc008> from fs register
with length(bytes) 2
```

- PEB アドレスの取得

```
Mem Read <BV32 0xc0081000> from
<BV32 0xc0080030> with length(bytes) 4
```

- ProcessHeap アドレスの取得

PEB+0x18 のアドレスを計算している

```
Mem Read <BV32
mem_c0081018_7_32{UNINITIALIZED}> from
<BV32 0xc0081018> with length(bytes) 4
```

- ForceFlags フラグの取得

ProcessHeap+0x10 のアドレスを計算している

```
Mem Read <BV32 mem_90_9_32{UNINITIALIZED}>
from <BV32 __add__((((0#16 .. reg_122_8_16)
& 0xffff) << 0x10), mem_c0081018_7_32,
0x10)> with length(bytes) 4
```

A.1.3 NTGlobalFlag フラグの確認

- FS レジスタからの読み込み

```
Reg Read <BV16 0xc008> from fs register
with length(bytes) 2
```

- PEB アドレスの取得

```
Mem Read <BV32 0xc0081000> from
<BV32 0xc0080030> with length(bytes) 4
```

- NTGlobalFlag フラグの取得

PEB+0x68 のアドレスを計算している

```
Mem Read <BV32
mem_f0021068_10_32{UNINITIALIZED}> from
<BV32 (((0#16 .. reg_122_8_16) & 0xffff)
<< 0x10) + 0xc0081068> with length(bytes) 4
```

A.2 実験 2 における angr の履歴

A.2.1 ファイル名を構成する各文字についての制約

次の条件式で<BV32 0x1>はメモリアドレス 0x1 を指しているため，メモリアドレス 0x1 から 1 バイトのデータをシンボル値 mem_1_11_8 に読み込んでいる．

```
Mem Read <BV8 mem_1_11_8{UNINITIALIZED}> from
<BV32 0x1> with length(bytes) 1
```

次に，読み込んだ 1 バイトが 112(='p') と一致する場合と異なる場合の条件を抽出している．

```
Constraint (<Bool mem_1_11_8 == 112>,)
Constraint (<Bool mem_1_11_8 != 112>,)
```

同様にして 2 文字目，3 文字目，…，8 文字目について満たすべき条件が抽出できる．

ファイル名の末尾である 8 文字目について，次の条件式でメモリアドレス 0x8 から 1 バイトのデータをシンボル値 mem_8_31_8 に読み込んでいる．

```
Mem Read <BV8 mem_8_31_8{UNINITIALIZED}> from
<BV32 0x8> with length(bytes) 1
```

次に，読み込んだ 1 バイトが 0(='\x00') と一致する場合と異なる場合の条件を抽出している．

```
Constraint (<Bool mem_8_31_8 == 0>,)
Constraint (<Bool mem_8_31_8 != 0>,)
```