動的解析と静的解析を連携したマルウェアのサンドボックス回避条件抽出

本多, 恵佑 / HONDA, Keisuke

(出版者 / Publisher)
法政大学大学院理工学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 理工学研究科編
(巻 / Volume)
65
(開始ページ / Start Page)
1
(終了ページ / End Page)
8
(発行年 / Year)
2024-03-24
(URL)
https://doi.org/10.15002/00030757

動的解析と静的解析を連携したマルウェアのサンドボックス回避条件抽出

EXTRACTION OF SANDBOX EVASION CONDITIONS FOR MALWARE BY COMBINING DYNAMIC AND STATIC ANALYSIS

本多恵佑
Keisuke HONDA
指導教員　金井敦

法政大学大学院理工学研究科応用情報工学専攻修士課程

In this study, we address the daily analysis of malware, particularly focusing on types that include functions to evade detection. Notably, we examine malware that can detect an analysis environment, such as a sandbox, and conceal its presence. We propose a simple and rapid method to extract the conditions under which such malware detects sandboxes and avoids analysis. This method mechanistically extracts evasion conditions by integrating dynamic binary instrumentation (dynamic analysis) and symbolic execution (static analysis). As a preliminary step to experiments using actual malware, we prepared pseudo-malware equipped with various evasion functions and evaluated them by extracting evasion conditions using the method proposed in this study. As a result, we succeeded in identifying evasion conditions for 5 out of 13 samples and obtained clues to evasion conditions for the other 6 samples. On the other hand, it became clear that there is room for improvement in the programs used for dynamic binary instrumentation and symbolic execution.
***Key Words** : Malware analysis, Sandbox evasion, Dynamic binary instrumentation, Symbolic execution*

１．序論

マルウェアと呼ばれる悪意のあるソフトウェアは，日々研究者や解析者によって解析が行われている．解析プロセスの一種である動的解析では，サンドボックスと呼ばれる使い捨ての仮想環境内でマルウェアを実行することにより，安全にマルウェアの振る舞いを分析することが可能である．一方で，こうした解析をされないための回避機能が実装されたマルウェアが存在する．具体的には，自身が実行された環境がサンドボックスかどうか確認を行い，該当した場合に強制終了する機能や，善良なソフトウェアであるかのように振る舞うといった機能を実装することがある．これにより解析に時間がかかるだけでなく，サンドボックスを活用したセキュリティ製品においても安全であると判断されてしまう恐れがある．サンドボックス以外にも，デバッガによる解析やメモリダンプによる解析など様々な解析を回避する機能を搭載したマルウェアがあり，これらは総じて回避型マルウェアと呼ばれている．Lorenzo Maffia らの調査[1]によると，2016 年から 2020 年にかけて収集した約 18 万ものマルウェアの内，40%以上が少なくとも 1 つの回避機能を搭載していることが示された．このことから，今後も新たな回避型マルウェアが生まれることが予測されるため，対策する必要があると考えられる．

マルウェアがサンドボックス環境を検知する方法は，大きく分けて 2 種類存在する．1 つ目は，VMware や VirtualBox といった仮想化ソフトウェアのシステム情報をマルウェアが確認する方法である．MAC アドレスや CPU 情報，稼働中のプロセスなどには仮想化した環境でしか見られない特徴が存在する．このような特徴を検出することでマルウェアが実行された環境が仮想環境であると判断することが可能である．2 つ目は，人間によって操作されていないことを推測する方法である．サンドボックスを用いた動的解析は一般的に自動化されている上に，一度に複数のマルウェアを解析するためにリソースも物理環境に比べて少ない傾向がある．このことから，マウスによる操作が極端に少ない場合や，画面解像度，ストレージサイズ，メモリサイズなどの情報を取得した際に明らかに低い値を示す場合はサンドボックスであると推測することが可能である．このようなサンドボックスを検知する手法は数多く存在し，日々新しい手法が生み出されている．

サンドボックス回避機能を持つマルウェアに対し，解

析者たちは独自のサンドボックスを構築し，実際の物理環境に近い環境を用意することで多くの回避機能を無効化している[2][3]．しかし，今後新たな回避機能を持ったマルウェアが生み出される中，回避されにくいサンドボックスを作り続けるには，最新のマルウェアがどのようにしてサンドボックスを検出して回避しているか，という条件を知る必要がある．静的解析でアセンブリ言語を解読することで，条件を特定することが可能ではあるが，膨大な量のアセンブリ言語を理解し分析するのは多くの時間と労力が必要である．

２．関連研究

窪優司らは，シンボリック実行を活用することでマルウェアが持つ解析環境検知機能を回避する条件を自動抽出する手法を提案している[4][5]．シンボリック実行は，プログラム解析技術の 1 つであり，プログラム内の特定のアドレスへ到達するための条件を導き出すことが可能である．シンボリック実行によって，環境検知機能の直前から検知されなかった場合に実行される命令アドレスへ到達する条件を抽出し，マルウェアがどのように環境検知を行っているかを明らかにするというものである．この研究ではデバッガ検知やサンドボックス環境検知，フック検知などの機能を備えたテストプログラムである pafish[6]に対して提案手法を適用した．その結果，pafish が持つ 55 個の環境検知機能のうち約 6 割の 34 個の検知条件を抽出することができた．一方でシンボリック実行を使用するにあたって，探索開始位置である開始アドレスと到達を目指す位置となる目標アドレスを指定する必要がある．そのため，マルウェアに適用する場合は目標となる悪性部分のアドレスをリバースエンジニアリングによって探し出さなければならないという課題が存在する．

西田雄亮は，動的解析と静的解析を活用した回避型マルウェアの回避コード抽出手法を提案している[7]．まず動的解析の 1 つである動的バイナリ計装を利用し，回避型マルウェアを仮想環境内で実行した際にコードセグメント中で通過した命令を明らかにする．仮想環境で無害な振る舞いをした場合，実行されたのは環境を検知するための関数と，無害な振る舞いをする関数であると推測できる．つまり，記録された実行済みの命令群は環境検知関数または無害な振る舞いをする関数であると言える．一方で，コードセグメント内で実行されなかった命令群には，他の環境検知関数と悪性な関数が含まれていると考えられる．記録されなかった環境検知関数は記録された環境検知関数と同様に，検知した際に無害な振る舞いをする関数を呼び出す．これを踏まえ，実行されなかった関数の中で無害な振る舞いをする関数に到達するものを静的解析によって抽出する．到達の可否については，JMP 命令や CALL 命令の引数となるジャンプ先アドレスを基に判断している．以上の手順により回避コードの抽出を試みている．また，提案手法のプロトタイプを作成し，擬似検体を用いて評価した結果，回避コードを 50¥%~60¥%抽出することに成功した．この研究では，動的解析時に仮想環境で実行された回避コードが抽出できない点と，回避コードごとに無害な振る舞いをする関数が存在した場合に回避コードを正しく抽出できない点が課題となっている．

本研究ではマルウェアのサンドボックス回避条件を簡単かつ短時間で抽出することを目指し，動的解析である動的バイナリ計装と静的解析であるシンボリック実行を連携した手法を提案する．

３．使用技術

３．１．動的バイナリ計装

動的バイナリ計装（DBI: Dynamic Binary Instrumentation）はプログラムの動的解析手法の 1 つである．プログラム中の関数をフックして入出力を監視することや，実行中の命令やデータに変更を加えること，事前に用意したコードを挿入することが可能である．本研究では，プログラム実行時に基本ブロックが呼び出されるたびに，その基本ブロックの情報を記録する．こうして得られた情報をカバレッジデータと呼び，実行された命令と実行されなかった命令の可視化に利用している．

３．２．シンボリック実行

シンボリック実行はプログラムの静的解析手法の 1 つである．入力する値をシンボル，すなわち変数として扱い，プログラムの解析によって実行可能なパスの抽出や，特定のパスに至るための入力値の条件を特定するものである．例として，図 1 に示すようなフローグラフに沿って実行される関数を持つプログラムがあったとする．

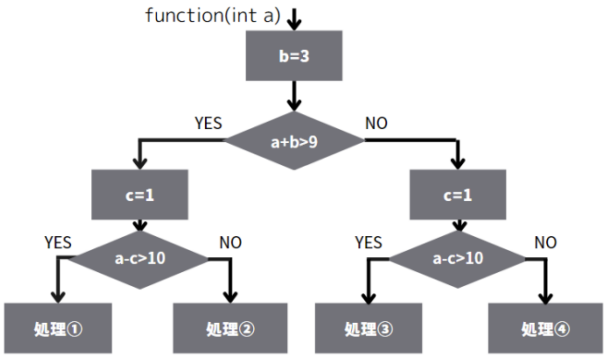

図 1　プログラム例

実際にこの関数が実行された際には，変数 a に具体的な整数の値が格納され，いずれか 1 つの経路を通ることとなる．一方シンボリック実行では実際に実行するのではなく，変数 a に格納される値をシンボル x と置き換え，関数内に存在する変数や分岐条件を収集しながら経路探索を行う．そして，目的のパスに到達する経路が明らか

になった段階で条件を整理する．これにより，図 1 中の処理①に到達するのは次の連立方程式を満たすシンボル x が存在した場合となることが判明する．

$$\begin{cases} b = 3 \\ x + b > 9 \\ c = 1 \\ x - c > 10 \end{cases} \quad (1)$$

制約を全て満たせるシンボル x の値が存在するかどうかを判定する問題のことを充足可能性問題（SAT: Satisfiability problem）と呼び，答えを導く手法を SAT ソルバと呼ぶ. SAT ソルバによって, そのパスに到達可能かどうか, そして到達可能な場合はシンボル x が具体的にどのような値を取る時かを求めることができる．今回の例では，処理①に到達するのは x ＞ 11 の時であることが明らかになる．一方で，処理③に到達するための解が存在しないため，処理③には到達不可能であることも判明する．シンボリック実行では前述した関連研究[6][7]で既に活用されており，本研究ではマルウェアが持つ悪性部分に到達するための条件抽出に使用する．

４． 提案手法

関連研究で紹介した文献[6][7]では，回避条件の抽出にはシンボリック実行は有効だがマルウェアの悪性部分のアドレスを手動で特定しなければならないという課題があった．また，回避コードの抽出を試みる文献[9]では，動的バイナリ計装により仮想環境で実行された命令を抽出する手法を用いていた．そこで，本研究ではこれらの手法を組み合わせ，動的バイナリ計装を活用することで悪性部分のアドレスを抽出し，シンボリック実行を用いてマルウェアがサンドボックスを回避する条件を抽出するという手法を提案する. 提案手法の概要を図 2 に示す.

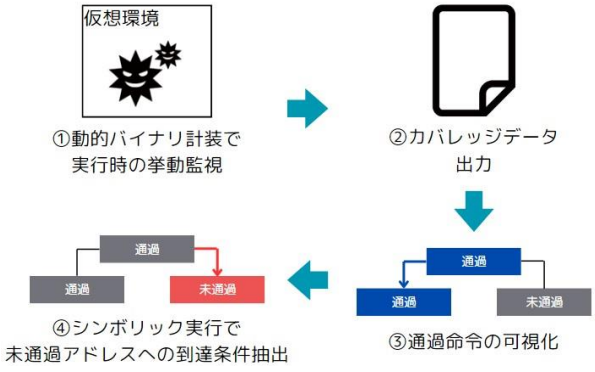

図２　提案手法の概要

まず，動的バイナリ計装を説明するにあたり，サンドボックス回避機能を備えたマルウェアの挙動を図 3 に示す.

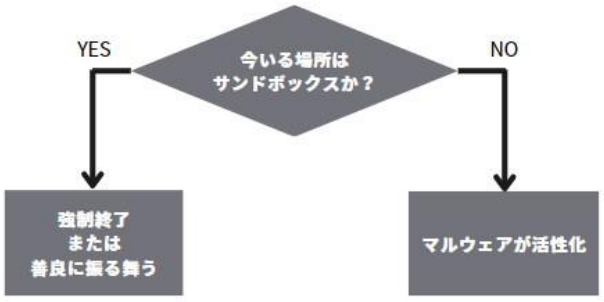

図３　サンドボックス回避の挙動

サンドボックス環境で実行した場合，マルウェアは活性化せずに悪性部分の隠蔽を試みる．つまり，実行されなかった基本ブロックにマルウェアの悪性部分が存在すると考えられる．そこで動的バイナリ計装を用い，サンドボックス環境で実行した際にどの命令が実行されたかを示すカバレッジデータを収集する．これにより，図 4 のように実行時に通過した基本ブロック，命令を可視化することが可能となる．

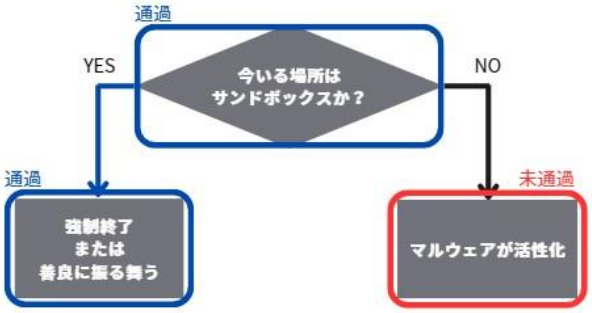

図４　動的バイナリ計装による通過命令の可視化

マルウェアが活性化した際に実行される基本ブロックの命令数は，前処理や例外処理が行われる基本ブロックで実行される命令数に比べて多くなることが予想される．したがって，実行時に通過しなかった基本ブロックが複数存在した場合は，命令数が最も多い基本ブロックにマルウェアの悪性部分が含まれると推測する．本研究では，動的バイナリ計装のフレームワークである Intel Pin[8]を採用しており，そのオープンソースモジュールである ddph[9]を利用することでカバレッジデータの収集を行う．サンドボックス環境で解析対象のプログラムを実行した際，対象のマルウェアが長時間終了せず，待機し続ける可能性がある．これを考慮し，基本ブロックごとに経過時間を確認し, 2 分経過していた場合にはその時点でのカバレッジデータを出力し，計測を終了するよう ddph を改良した．こうして得られたカバレッジデータは，リバースエンジニアリングツールの Ghidra[10]とそのオープンソースプラグインである Dragon Dance[9]を用いることでフローグラフ上に可視化する．

次に，シンボリック実行によって悪性部分と判断した

アドレスに到達するための条件を抽出する．シンボリック実行では経路探索を開始する位置である開始アドレスと，到達したいアドレスである目標アドレスを指定する必要がある．目標アドレスについては，前述の通り動的バイナリ計装によって求める．一方で，開始アドレスは main 関数のアドレスとした．サンドボックス環境かどうかを確認し，それによって分岐を行う命令はマルウェアの悪性部分よりも前である必要がある．よって，main 関数の前半に分岐命令が配置されると推測されるため，開始アドレスは main 関数のアドレスに設定する. main 関数のアドレスは，リバースエンジニアリングツールから特定のバイナリパターンを検索することで見つけることができる. 例として, 本研究で扱う x86 版 PE ファイルでは，CALL 命令で main 関数が呼び出される前に引数としてスタックに ImageBase が PUSH される. ImageBase は実行ファイルがロードされるメモリ上のアドレスである．PUSH 命令のオペコードが 0x68 であるため，ImageBase が 0x00400000 を取る場合はバイナリパターンは“68 00 00 40 00”となる．また，main 関数を呼び出す CALL 命令のオペコードは 0xe8 であることから, 図 5 に示すように“68 00 00 40 00 e8”のバイナリパターンを検索することで，直後に存在する main 関数のアドレスを特定することができる．

```
00401420 50              PUSH    EAX
00401421 57              PUSH    EDI
00401422 68 00 00 40 00  PUSH    IMAGE_DOS_HEADER_00400000
00401427 e8 e4 fb ff ff  CALL    FUN_00401010
0040142c 8b f0           MOV     ESI,EAX
0040142e e8 3c 05 00 00  CALL    FUN_0040196f
```

図 5　main 関数のアドレス特定

main 関数の呼び出し方法はコンパイラやリンカのバージョン，オプションにより変化するが，いくつかのパターンと照らし合わせることで特定が可能である[11]. 本研究では，シンボリック実行フレームワークとして，angr[12]を採用している. angr は python3 のライブラリとして提供されている．開始アドレスから目標アドレスに到達する条件に加え，メモリ値へのアクセスログと関数の呼び出し記録を出力するスクリプトを作成した．

以上の提案手法を総括すると，まず仮想環境内で対象のマルウェアを実行し，その際に通過した命令と通過しなかった命令を動的バイナリ計装によって明らかにする. その後，通過しなかった基本ブロックの中で最も命令数の多い基本ブロックを悪性部分であると推測する．最後に，それを基にシンボリック実行をすることでマルウェアの活性化条件を出力する．マルウェアの活性化条件が判明するということは，その裏である回避条件が判明するため，以上をもって回避条件の抽出を試みる．

５． 評価

５．１．擬似マルウェア

提案手法の評価にあたり，サンドボックス回避機能について紹介されている文献[13][14]を参考に以下のサンドボックス解析回避機能を搭載した擬似的なマルウェアを作成した．擬似マルウェアであるため，活性化した場合は悪性な挙動ではなく「マルウェアが活性化しました」というメッセージボックスが画面上に表示されるようにしている．なお，実行ファイルは WindowsPE の EXE ファイルとしてコンパイルされており，特に明記されていないものは x86 版のプログラムである．

- 日付確認
  現在の日付情報を取得し，特定の日付でのみ活性化する．それ以外の日付では起動後直ちに終了する．

- 一定時間待機
  一定時間待機後に活性化する．これは自動サンドボックス解析における，タイムアウトによる解析終了を狙った回避機能となる．

- 仮想環境ファイル検知
  仮想環境特有のファイルを探し，検知しなければ活性化する．検知した場合は直ちに終了する．x86 版では存在の有無に関わらず検知しないため x64 版を採用した．

- アクティブウィンドウ監視
  アクティブになっているウィンドウハンドルが変化したら活性化する．変化しない場合は待機し続ける．自動サンドボックス解析では人間による操作が行われないため，そうした特徴の検知を試みる回避機能である．

- MAC アドレス確認
  ネットワークアダプタの MAC アドレス情報を取得し，仮想環境特有のパターンが現れた場合に直ちに終了する．VirtualBox であれば，MAC アドレスの前半は必ず 08-00-27 となる．このパターンに当てはまらない場合は活性化する．

- ストレージサイズ確認
  ストレージサイズを確認し，一定の容量を下回った際には仮想環境と判断し直ちに終了する．一定以上の容量があれば物理環境と判断し，活性化される．

- メモリサイズ確認
  ストレージサイズと同じ要領でメモリサイズを確認し，仮想環境か判断する．

- ハイパーバイザーベンダ ID の取得
  CPUID命令によってハイパーバイザーベンダ IDが取得できた場合は直ちに終了し，取得できなければ活性化する．ID の例として，VirtualBox では「VBoxVBoxVBox」という文字列が取得できる．

- 稼働プロセスの確認
  仮想環境では物理環境では見られないプロセスが稼働している．VirtualBox では VBoxService.exe や VBoxTray.exe が該当する．これらを検知した際には強制終了し，検知しなければ活性化する．

- 論理プロセッサ数確認
  論理プロセッサ数を確認し，一定の数量を下回った際には仮想環境と判断し直ちに終了する．一定以上の数量があれば物理環境と判断し，活性化する．

- ゴミ箱内ファイル数確認
  デスクトップ上に存在するゴミ箱フォルダを確認し，ファイル数が一定数以下であった場合に仮想環境と判断し直ちに終了する．ファイル数一定数以上あれば物理環境と判断し，活性化する．サンドボックスはユーザが一般的な用途で使用せず，解析する度にリセットされる．そのため，ゴミ箱フォルダ内にはファイルがほとんど存在しないことを推測した回避機能である．

- 起動後経過時間確認
  端末が起動してからの経過時間を取得し，起動直後であることが判明した場合に直ちに終了する．起動後長時間が経過していれば活性化する．サンドボックスは解析のためだけに起動されるため，起動直後に実行された場合には解析環境であることが推測できる．

- 画面解像度確認
  縦方向のピクセル数と横方向のピクセル数を取得し，一定の数値を下回った際には仮想環境と判断し直ちに終了する．一定の数値を上回れば物理環境と判断し，活性化する．

５．２．実験環境

本研究では，動的バイナリ計装を行うための仮想環境として，Oracle VM VirtualBox 上に構築した 64bit 版 OS の Windows10 を使用する．ストレージサイズは 50GB，メモリサイズは 2GB とした．また，カバレッジデータの可視化とシンボリック実行を行うための環境として，同様に VirtualBox 上に構築した 64bit 版 OS の Ubuntu22 を利用している．各種ソフトウェアのバージョン情報を表１に示す．

表１　各種ソフトウェアのバージョン情報

ソフトウェア	バージョン情報
Oracle VM Virtual Box	7.0.8
仮想 Windows10 Pro 64bit	22H2
仮想 Ubuntu22 64bit	22.04.3 LTS
Intel Pin	3.7
Ghidra	9.0.2
Dragon Dance	0.2.2
angr	9.2.5

５．３．実験１

5.1 で述べた擬似マルウェアに対し，以下の手順に従い回避条件が抽出できるか実験を行った．なお，以下の手順のうち１が動的解析部分とし，3.4.5 が静的解析部分とする．

1. 仮想環境の Windows10 上にて，各擬似マルウェアを実行し，動的バイナリ計装を使用してカバレッジデータの収集を行った．なお，2 分以上経過してもプログラムが終了しない場合には強制的にプロセスを終了した．

2. 出力されたカバレッジデータを，対象の擬似マルウェアと共に仮想環境の Ubuntu に移した．

3. Ghidra に擬似マルウェアとカバレッジデータを読み込ませ，フローグラフに実行された命令群を可視化した．この時，バイナリパターンの検索機能を使用して main 関数のアドレスを特定し，記録した．また，動的解析時に実行されなかった基本ブロックの中で最も命令数の多いもののアドレスを記録した．

4. 得られたアドレス情報を基に擬似マルウェア，開始アドレス，目標アドレスを python スクリプト上で指定し，angr によるシンボリック実行を行った．

5. シンボリック実行によって出力された条件とログ情報から，擬似マルウェアがサンドボックス解析を回避する条件が得られるか検証した．

５．４．実験２

実験１の後，回避条件抽出に成功した検体を１つ選び，難読化処理を行った後に実験 1 と同様の手順で条件抽出ができるか実験を行った．なお，検体の難読化にはフリーウェアである UPX[15]を利用し，圧縮レベルを 2,5,8,b(最大)の 4 種類作成した．圧縮レベルは数字が大きいほど圧縮率が高くなり，ファイルサイズが小さくな

る．また，圧縮レベルの異なる 4 種の検体について，難読化を解除した後に再び実験 1 と同じ手順で検証を行った．

６． 実験結果

６．１．実験１の結果概要

実験結果を表２に示す．回避条件を抽出できたものは○，抽出できなかったが回避機能の手がかりを得られたものを△，抽出できず，手がかりも得られなかったものは×で示されている．

表２　実験結果概要

擬似マルウェア	回避条件抽出の可否
日付確認	○
一定時間待機	×
仮想環境ファイル検知	○
アクティブウィンドウ監視	○
MAC アドレス確認	△
ストレージサイズ確認	△
メモリサイズ確認	△
ハイパーバイザーベンダ ID の取得	×
稼働プロセスの確認	△
論理プロセッサ数確認	△
ゴミ箱内ファイル数確認	△
起動後経過時間確認	○
画面解像度確認	○

６．２．実験１ － 成功した検体

回避条件が抽出できた例として，日付確認を行う検体を挙げる．動的解析部分で得られたカバレッジデータを可視化すると，図６に示す結果が得られた．

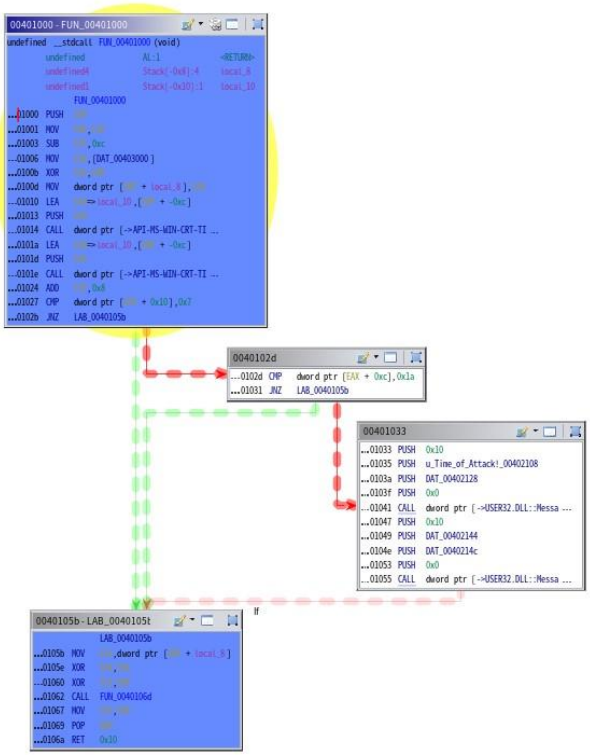

図６　日付確認を行う検体のカバレッジデータ

図６では，動的解析時に実行された基本ブロックが青くハイライトされた．実行されなかった基本ブロックの中で最も命令数の基本ブロックのアドレスが 0x401033，main 関数のアドレスが 0x401000 であり，これらを基にシンボリック実行を行った結果の関連個所を図７に示す．

```
1   Bool mem_10_3_32{UNINITIALIZED} == 0x7,
2   Bool mem_c_4_32{UNINITIALIZED} == 0x1a
```

図７　日付確認を行う検体のシンボリック実行結果

図７の 1 行目では，仮想メモリの 0x10 番地のアドレスに置かれた 32bit 長の値が 0x7 であることが目標アドレスに到達するための条件の 1 つであることを示している．次に，angr が出力したログの内，シンボル化された 2 つの変数に関連する箇所を図８に示す．

```
1   <SimActionData 0x401027:8 mem/read: <BV32
        unconstrained_ret__localtime64_2_32{
        UNINITIALIZED} + 0x10> ---->> <BV32
        mem_10_3_32{UNINITIALIZED}>>,
2   <SimActionData 0x40102d:3 mem/read: <BV32
        unconstrained_ret__localtime64_2_32{
        UNINITIALIZED} + 0xc> ---->> <BV32
        mem_c_4_32{UNINITIALIZED}>>
```

図８　日付確認を行う検体のシンボリック実行ログ

図８では，図７で示した 2 つの変数が参照された際の記録を示しており，どちらの値も localtime64 という文字列が含まれたアドレスに読み込まれている．localtime64 は日時情報である time 値を tm 構造体に変換し，その tm 構造体のポインタを返す関数である．tm 構造体では，秒，分，時，日，月の順で int 型の変数が用意されている．int 型の変数の長さが 4byte であることから，“unconstrained_ret__localtime64_2_32{UNINITIALIZED}+0x10”は月を，“unconstrained_ret__localtime64_2_32{UNINITIALIZED}+0xc”は日にちを表す整数値が格納される．tm 構造体では月を示す値が 0 から始まることから，図７中の 0x7 は 8 月，0x1a は 26 日を示していることが明らかになった．よって，日付が回避条件であることが判明した．本検体以外においても，成功した検体は同様の方法で回避条件を得られた．

６．３．実験１ － 失敗したが手がかりが得られた検体

回避機能の手がかりが得られた例として，メモリサイズ確認を行う検体を挙げる．この検体では，シンボリック実行時に出力された結果が“Bool mem_7ffefedc_2_32{UNINITIALIZED¥} > 0x0”であった．これは仮想メモリ上のアドレス 0x7ffefedc に置かれた変数の値が 0 より大きいときにマルウェアが活性化することを示している．一方で，angr が出力したログを調査してもこの変数が何

によって決まるかが判明せず，回避条件の特定には至らなかった．しかし，アドレス 0x7ffefedc が参照された際に実行された命令のアドレス情報や，直前に“GlobalMemoryStatusEx”関数が呼び出されていたことが記録されていたため，メモリに関する情報が関連しているという手がかりが得られた．

６．４．実験１ － 失敗した検体

一定時間待機を行う検体では，動的解析時に 2 分経過してもプログラムが終了しなかったため，手動で強制終了した．その結果，カバレッジデータが出力されなかったため，回避条件の抽出には至らなかった．

ハイパーバイザーベンダ ID の取得による回避機能を搭載した検体では，シンボリック実行による出力は得られたものの，その結果が条件なしで目標アドレスに到達可能であることを示していた．しかし，動的解析時には活性化しなかったことから，回避条件の抽出は失敗となった．

６．５．実験２の結果

実験 1 で回避条件の抽出に成功した，日付確認を行う検体に対し 4 種類の圧縮レベルで難読化処理をし，実験を行った．日付確認を行う検体のファイルサイズは 102KB であり，圧縮後は最も低い圧縮レベルの検体が 60KB，最も高い圧縮レベルの検体が 58KB となった．実験 1 と同様の手順で回避条件の抽出を試みた結果，カバレッジデータの取得には成功したが，全ての検体においてその後のカバレッジデータの可視化に失敗した．一方で，難読化を解除した後に同様の手順で実験を行ったところ，4 種類すべてにおいて回避条件の抽出に成功した．なお, 抽出結果については 6.2 で示した日付確認の検体の結果と同じであったためここでは省略する．

７．分析と考察

７．１．実験１で抽出に成功した検体について

実験で使用した 13 個の検体のうち，5 個の検体で回避条件の抽出が成功した．これらは全て angr の出力結果やログに，WindowsAPI 関数や C 言語の標準ライブラリに含まれる関数の名前が記載された論理式が存在したため容易に特定することができた．一方で，こうした既存の関数を極力使わずに実装された回避機能を搭載したマルウェアに対して実施する場合には，ログ解析が長時間かかると推測される．また，今回条件抽出に成功した検体全てにおいて，手動でのログ解析が必要であったため，今後は angr のログ出力設定の見直しとログ解析の効率化を検討する必要がある．また, 今回使用した検体は全て，実行されなかった基本ブロックの内，最も命令数の多い基本ブロック内に活性化時の挙動が含まれていた．しかし，ソースコード内に本来の挙動とは全く関係のないジャンクコードを挿入することで，シンボリック実行時の目標アドレスを誤って指定してしまう恐れがある．そのため，マルウェアの悪性部分を特定するより良い方法を模索し，本提案手法と組み合わせることでさらなる改良ができると考えられる．

７．２．実験１で手がかりが得られた検体について

実験で使用した 13 個の検体のうち，6 個の検体では回避条件の抽出には失敗したが，回避機能で使用される関数や，分岐に関わる命令のアドレスといった手がかりを得られた．本提案手法のみでの回避条件特定には至らなかったものの，得られた手がかりを基にリバースエンジニアリングやデバッグによる解析に役立てることができるのではないかと考えられる．一方で，angr のログ出力設定の見直しや，より詳細なログ解析によって条件特定につながる可能性があるため，引き続き手法の改善に取り組むべきである．

７．３．実験１で抽出に失敗した検体について

一定時間待機を行う検体について，カバレッジデータが出力されなかった原因を調査した結果，Sleep 関数によるものではないかという結論に至った．今回使用した ddph を改良したプログラムでは，基本ブロックを 1 つ通過するごとに経過時間を確認し, 2 分経過していた場合にプログラムを終了してその時点でのカバレッジデータを出力する．しかし，Sleep 関数が呼び出された際には定められた時間プログラムが停止する．そのため，動的解析時に基本ブロックを通過しきることはなく，Sleep 関数のアドレスで長時間停止していたことがカバレッジデータの取得に失敗したと考えられる．この検体では例外的に分岐命令が存在せず，シンボリック実行を行う前にフローグラフを確認した時点で Sleep 関数の存在が確認できるため，カバレッジデータの取得に失敗した場合でもフローグラフを確認することで回避条件を特定できると考えられる．

ハイパーバイザーベンダ ID の取得を行う検体では，回避条件が存在しないという誤った結果が抽出された．この原因として，angr の最適化機能によるものではないかと考えられる．シンボリック実行時に全ての関数内の処理を分析すると多大な時間を要する．angr はこれを避けるために，デフォルトで一部のライブラリ関数の動作を変更し，予想される戻り値を返すだけの関数に置き換えている．よって，実験時には CPUID 命令で得られるハイパーバイザーベンダ ID の文字列がシンボル化されず，ID の取得に失敗する命令に変更されたと考えられる．その結果，条件なしで活性化するという出力が得られたと推測される．これは，angr の SimProcedure 機能により実装されている．一方で，SimProcedure は必要に応じてカスタマイズすることが可能であるため，CPUID 命令を使った回避機能のような既知の手法については対応が可能であると思われる．また，未知の回避機能に対して同様に

条件なしで到達可能という結果になった場合は，デフォルトで SimProcedure が有効になっている関数を調査することで特定時間を短縮できるのではないかと考えられる．

７．４．実験２について

難読化処理を行った検体全てにおいてカバレッジデータ読み込みに失敗した原因として，動的解析時に収集したカバレッジデータと Ghidra が解析し作成されたフローグラフが異なる構造をしていたためではないかと考えられる．動的解析時には，実際に擬似マルウェアが実行されるため，圧縮されたプログラムコードが解凍され，元のプログラムの命令が順に実行されることとなる．よって，カバレッジデータに記録されるものに元のプログラムの命令アドレスが含まれる．一方で，Ghidra からフローグラフ表示を行った際には，表示されるのは圧縮されたプログラムを解凍するための命令群と，解凍後に展開された命令群にジャンプするための命令である．よって，Ghidra では解凍後の命令群が表示されないため，カバレッジデータとの齟齬が生じてしまい，可視化に失敗したと考えられる．

マルウェアは難読化処理を施していることが多く，難読化に使用されるソフトウェアも様々であることから，別途対策をとる必要がある．一方で，今回実験で行ったように難読化の解除が容易な場合には，解除した後に本提案手法を用いることで回避条件の抽出が期待できる．

８．結論

本論文では動的解析である動的バイナリ計装と静的解析であるシンボリック実行を連携することで，サンドボックス回避機能を持ったマルウェアの回避条件を効率的に抽出する手法を提案した．そして，異なる回避機能を搭載した擬似的なマルウェアを複数作成し，それらに対し提案手法を適用することで評価を行った．検証の結果，13 種類の検体のうち 5 種類の回避条件の特定に成功し，他 6 種類では回避条件の手がかりが得られた．このことから，提案手法は従来のリバースエンジニアリングによる手作業でのコード解析を効率化できると考えられる．一方で，第８章の分析と考察で述べた通り，多くの課題が見つかった．そのため，今後はそれらを基に更なる改良を行う必要がある．また，サンドボックス回避手法やや解析妨害機能は今回扱った手法以外にも数多く存在するため，それらに対して検証と改善を繰り返し，最終的には実際のマルウェアへの有効性を検証する必要がある．

参考文献

[1] Lorenzo Maffia，Dario Nisi，Platon Kotzias，Giovanni Lagorio，Simone Aonzo and Davide Balzarotti，“Longitudinal Study of the Prevalence of Malware Evasive Techniques”，arXiv:2112.11289 [cs.CR]，2021.

[2] Trend Micro，“巧妙なマルウェアに対抗する最先端のサンドボックス技術”, トレンドマイクロセキュリティブログ，2017，https://blog.trendmicro.co.jp/archives/14720，（参照 2023-11-26）．

[3] Esmid Idrizovic，Bob Jung，Daniel Raygoza and Sean Hughes，“Navigating the Vast Ocean of Sandbox Evasions，UNIT 42 by palo alto networks”，2022，https://unit42.paloaltonetworks.com/sandbox-evasion-memory-detection/，（参照 2023-11-26）．

[4] 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”, 研究報告コンピュータセキュリティ(CSEC)，vol.2018-CSEC-82，no.39，pp.1-8，2018.

[5] 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環境検知機能の回避条件自動抽出の研究”, コンピュータセキュリティシンポジウム 2018 論文集，vol.2018，no.2，pp.770-777，2018.

[6] Alberto Ortega，pafish: Pafish is a testing tool that uses different techniques to detect virtual machines and malware analysis environments in the same way that malware families do，GitHub，https://github.com/a0rtega/pafish,（参照 2023-12-03）.

[7] 西田 雄亮，“回避型マルウェア解析のための回避コード抽出に関する研究”, 奈良先端科学技術大学院大学，奈良先端科学技術大学院大学先端科学技術研究科修士論文；43891，NAIST Digital Library，Available from: https://library.naist.jp/opac/volume/227914, 2020,（参照 2023-12-03）．

[8] Intel，“Pin a dynamic binary Instrumentation tool”，https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html，（参照 2023-10-26）．

[9] Oguz Kartal，“Dragon Dance: Binary code coverage visualizer and manipulator plugin for Ghidra”, GitHub, https://github.com/0ffffffffh/dragondance，（参照 2023-10-20）．

[10] National Security Agency，“Ghidra”，https://ghidra-sre.org/，（参照 2023-12-02）．

[11] 中島 将太, 小竹 泰一, 原 弘明, 川畑 公平, “リバースエンジニアリングツール Ghidra 実践ガイド”，マイナビ出版，pp.75-p76，2020.

[12] Shoshitaishvili Yan，Wang Ruoyu，Salls Christopher，Stephens Nick, Polino Mario, Dutcher Audrey, Grosen John，Feng Siji，Hauser Christophe，Kruegel Christopher，Vigna Giovanni，“SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”，IEEE Symposium on Security and Privacy，pp. 138-157，2016.

[13] Thomas Roccia，Jean-Pierre LESUEUR，“Sandbox Evasion - Unprotect Project”，https://unprotect.it/category/sandbox-evasion/，（参照 2023-12-03）．

[14] Shaul Vilkomir-Preisman，“Malware Evasion Techniques Part 2: Anti-Virtual Machines”，2019，https://www.deepinstinct.com/blog/malware-evasion-techniques-part-2-anti-vm-blog,（参照 2023-12-03）.

[15] Markus F.X.J. Oberhumer，László Molnár，John F. Reiser，“UPX: the Ultimate Packer for eXecutables”，https://upx.github.io/，（参照 2024-2-12）．