デバッガ検知回避および不要なループからの脱出により例外発生マルウェアを実行継続させる動的解析手法

神谷 直輝[1,a)]　山根 一真[1]　掛井 将平[1]　齋藤 彰一[1]

概要：マルウェアを動作させ，実際の挙動を解析する動的解析は，マルウェアの攻撃の方法や流れなどを解析者が把握するために行われる．しかし，マルウェア開発者の意図しないバグの混入などにより解析時に例外が生じ，実行が中断するという問題がある．動的解析のこのような状況に対して，大山らは ExMod を提案した．これは，動的解析ソフトウェアの Cuckoo Sandbox 上でマルウェアを実行する際，例外発生命令のスキップや，例外発生に起因する無意味なループからの脱出を行い，例外発生後の実行と解析の継続を実現した．ところが，マルウェアに実装された例外処理の実行の有無で，デバッガを検知する手法が存在する．ExMod は先述の動作をマルウェアの例外処理に代わって実行するため，デバッガの誤検知を引き起こし，マルウェアが耐解析用の偽の処理を行う恐れがある．また，不要なループからの脱出操作は，操作内容やタイミングが噛み合わず，ループから脱出できないケースを確認した．そこで本稿では ExMod の処理を検討し，例外発生命令の，デバッガ検知を回避したスキップと，不要なループからの脱出を行う新たな手法を提案する．

キーワード：動的解析，Windows マルウェア，例外処理，強制実行，サンドボックス

Methods for Dynamic Analysis to Keep Exception-raising Malware Running by Avoiding Debugger Detection and Escaping from Unnecessary Loops

Naoki Kamiya[1,a)]　Kazuma Yamane[1]　Shohei Kakei[1]　Shoichi Saito[1]

Abstract: Analysts use dynamic analysis to understand the behavior of malware. However, there is a problem with the analysis being interrupted due to exceptions being raised during execution. To overcome this problem, Oyama et al. proposed ExMod. When executing malware in Cuckoo Sandbox, ExMod skips instructions that raise exceptions and escapes from meaningless loops caused by exceptions to keep the malware running. However, there are methods to detect a debugger based on whether or not the exception handler is executed. The above behavior of ExMod triggers debugger detection because it is executed instead of the malware′s exception handler. In addition, we observed cases in which operations to escape from unnecessary loops could not escape. In this paper, we propose new methods for skipping instructions that raise an exception, avoiding debugger detection, and escaping needless loops.

Keywords: Dynamic Analysis, Windows Malware, Exception Handling, Forced Execution, Sandbox

1. はじめに

悪意のある動作を行うプログラム，いわゆるマルウェアが，企業や公共機関を攻撃する事例が後を絶たない [1]．こうした脅威に対策を講じるためには，マルウェアの解析を行い，攻撃者が用いる手口への深い理解が必要である．マルウェアの解析手法には，検体を実行し，ファイルやレジストリに対する変更，呼び出される API などを調べる動的

[1] 名古屋工業大学
Nagoya Institute of Technology
[a)] n.kamiya.852@nitech.jp

解析 [2] が存在する．解析者は動的解析により，マルウェアが感染端末に影響を及ぼす流れを把握できる．

動的解析には例外の発生によりマルウェアの実行が中断し，中断地点から先の挙動を解析できない問題がある．例外とは，通常の処理の中でプログラムの実行継続を阻害する異常な事象である．そのため，OS は例外が生じるとプログラムの実行を終了させる．プログラマは例外が生じる場合においてもプログラムを正常に実行継続させるべく，例外への対処，いわゆる例外処理を，例外が生じた際に実行される例外ハンドラ内に定義する．

しかし，マルウェア開発者の見落としにより，例外が生じうるコードに対して例外処理が施されない場合がある [3]．例外処理が施されていないコードの実行により例外が発生すると，解析環境上の OS がマルウェアの実行を中断するため，その先に潜む悪意のある動作を解析者が把握できない．そこで，動的解析において例外発生マルウェアを実行継続させる手法が提案されている [4][5][6]．マルウェアの実行中に例外が生じると，マルウェアに代わり独自の例外処理を行う．これにより，例外処理が施されていないコード部の実行によって例外が生じた場合の OS による実行中断を防ぎ，例外発生命令に続く処理からの実行継続を実現している．

一方，例外処理が施されているコード部で例外が生じて独自の例外処理が行われる場合，マルウェアの例外ハンドラの実行が阻害される．マルウェア開発者は解析者によるデバッガの使用を検知する目的で例外ハンドラをマルウェアに実装する場合がある [7]．マルウェア開発者は例外が必ず発生するコードと，デバッガ検知用の例外ハンドラおよびデバッガの有無を表す変数をマルウェア内に用意する．例外ハンドラには当該変数の値を書き換える処理を定義する．そして，例外ハンドラの実行後の通常の処理で変数の書き換えの有無を確認してデバッガ検知を行う．デバッガがアタッチした状態で例外が発生すると OS は例外の発生をデバッガに通知し，アプリケーションに実装されている例外ハンドラを実行しない [8]．したがって，変数の値が初期値から変わらないため，マルウェアはデバッガがアタッチしていると判断できる．このようなデバッガ検知を行うマルウェアに対して独自に例外処理を行うと，例外ハンドラの実行が阻害されるためマルウェアがデバッガを誤検知する恐れがある．また，ループ内で例外が発生する際に例外発生命令に続く処理から実行を再開させると，ループが実行継続を妨げる場合がある．本論文では，このようなループを不要なループと呼ぶ．3.3 節で示すように，不要なループはループ内の処理が正常に行われない．一般に動的解析は制限時間を設けて行われる [9] ため，解析の必要がない処理は実行を省略すべきである．

マルウェアを例外発生後も滞りなく実行継続させるためにはこれらの問題に対応する必要がある．本論文は例外発生マルウェアを実行継続させるために講じなければならない対策について議論を行い，問題に対応する新たな実行継続手法を提案する．

以下，2 章で動的解析において例外発生マルウェアの実行を継続させる関連研究について説明する．3 章では，ベンチマークプログラムやマルウェアを用いた関連研究の評価を行い，例外発生マルウェアの実行継続手法に求められる要件を検討する．4 章では，3 章での検討を踏まえた提案手法について説明し，5 章で提案手法の評価を行う．最後に，6 章でまとめを述べる．

2. 関連研究

マルウェアの動的解析時の例外発生問題に対応するため，マルウェアの実行中断を防ぎ，実行継続を行う研究について述べる．

2.1 X-Force

例外発生マルウェアの実行継続に関する代表的な研究に X-Force[4] がある．X-Force は未初期化のポインタへのアクセスで例外が発生すると，オンデマンドにメモリの確保を行い，ポインタを利用可能な状態にした上で実行を継続させる．ところが，このオンデマンドなメモリ割り当ては，きわめて大きなオーバヘッドが発生する．関連するすべてのポインタ変数を実行中に追跡して更新を行うため，未適用システムと比較して最大 473 倍の実行時間を要する．後継研究の PMP[5] では，このメモリ割り当ての改良により，X-Force と比較して 84 倍の高速化を実現したものの，依然としてオーバヘッドが大きく，この問題を十分に解決できてはいない．

2.2 ExMod

大山らは，動的解析ソフトウェアの Cuckoo Sandbox[10] 上で，例外発生マルウェアを実行継続させる ExMod[6] を提案している．ExMod は例外が発生する際，例外発生命令の次の命令から，あるいは，実行中の関数の呼び出し元から処理を再開させて，マルウェアを実行継続する．我々は ExMod を，X-Force や PMP 程のオーバヘッドを要さず，マルウェアの例外発生地点以降に潜む動作を明らかにできる手法として注目している．以下，本節では ExMod の動作について説明する．なお，全体像を**図 1** に示す．OS によって探索される例外ハンドラは 3.1 節で説明する．

Windows は例外が発生すると，当該例外を処理可能な例外ハンドラの探索と実行を行う（図 1 の Step 3, 3-a）．この一連の処理を行う Windows API を RtlDispatchException と呼ぶ．また，例外ハンドラの探索と実行の結果，アプリケーションを終了させるべきだと判断した場合，OS は NtTerminateProcess という Windows API を呼び出し，実行中のアプリケーションを終了させる（図 1 の Step 3-b）．

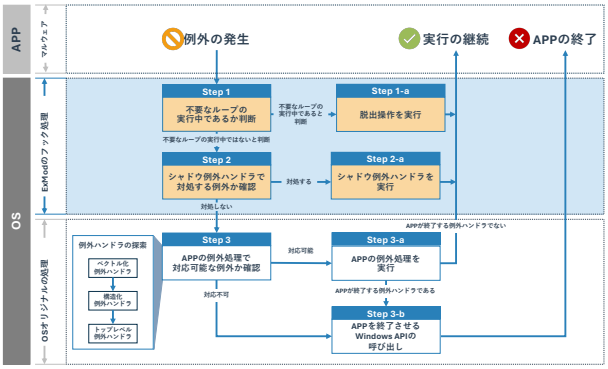

図 1　ExMod の全体像
Fig. 1　Overview of ExMod.

Cuckoo Sandbox は Windows API の処理の前に呼び出しを記録するための処理を追加する．このような，プログラム中の特定の箇所へのコードの追加をフックと呼ぶ．ExMod は RtlDispatchException へのフックコードを改変し，シャドウ例外ハンドラという，例外発生命令の後続処理から実行を継続させる独自の例外処理と，不要なループの実行を検知し，当該ループから脱出させる脱出操作を提案実装している．これらは，図 1 の ExMod のフック処理に該当する．不要なループの実行中であると判断（図 1 の Step 1）すると，脱出操作を行い不要なループから脱出させる（図 1 の Step 1-a）．不要なループの実行中でないと判断（図 1 の Step 1）すると，例外の種類を確認し（図 1 の Step 2），処理すべき種類の例外であればシャドウ例外ハンドラが例外処理を行う（図 1 の Step 2-a）．脱出操作とシャドウ例外ハンドラがいずれも実行されない場合，例外処理をフックコードに続く本来の RtlDispatchException の処理（図 1 の Step 3, 3-a, 3-b）に移譲する．

2.2.1　後続処理からの実行継続

図 1 の Step 2 において，処理する例外の種類とシャドウ例外ハンドラの処理内容を**表 1** に示す．これら発生頻度の高い 5 種類の例外が発生するとシャドウ例外ハンドラが実行を継続させ，例外ハンドラがマルウェアに実装されていない場合の OS による実行中断を防止する．

2.2.2　不要なループからの脱出

図 1 の Step 1 において，ExMod はマルウェアの実行中に以下の 2 つの条件を同時に満たす場合，不要なループの実行中であると判断し，ループから脱出するための操作を行う．

- 例外発生地点が 5 ヵ所以下に集中している
- 実行開始から通しで数えて 512 回以上例外が発生している

なお，不要なループから脱出するための操作は当該ループからの脱出に失敗し，ループを反復するたび，以下の順に実行される．

(1) 例外発生命令のスキップを行い実行再開

表 1　シャドウ例外ハンドラの処理
Table 1　Handling of Shadow Exception Handler.

例外発生命令の直後の命令から実行再開
EXCEPTION_ACCESS_VIOLATION
EXCEPTION_INT_DIVIDE_BY_ZERO
実行中の関数からのリターン
EXCEPTION_PRIV_INSTRUCTION
EXCEPTION_ILLEGAL_INSTRUCTION
EXCEPTION_STACK_OVERFLOW

(2) EFLAGS レジスタのビット反転を行い実行再開
(3) ECX レジスタの値を 0 に書き換えて実行再開
(4) 実行中の関数からのリターン
(5) シャドウ例外ハンドラに例外処理を移譲
(6) マルウェアの例外ハンドラに例外処理を移譲
(7) プログラムカウンタへランダム値を加算して実行再開

3.　例外発生マルウェアの実行継続に必要な対策

ExMod がシャドウ例外ハンドラを実行すると，マルウェアの例外ハンドラはバイパスされて実行されない．このため，例外処理を用いたマルウェアのデバッガ検知を誤って引き起こす恐れがある．この場合，マルウェアが解析に備えたダミーの処理を実行する恐れがあるため，デバッガの誤検知は望ましくない．また，大山らは不要なループの検知と脱出操作の正確性に課題があると述べている．

例外処理を用いたデバッガ検知を行うプログラムと実際のマルウェアを使用して ExMod の評価を行い，例外発生マルウェアを実行継続させる手法に求められる要件を検討する．評価環境は，ホストマシンは Intel Core i9-10900X，メモリ 256GB，Cuckoo ホスト VM は Ubuntu 20.04，物理コア 4，メモリ 6GB，Cuckoo ゲスト VM は Windows 7 Professional，物理コア 1，メモリ 2GB とした．

評価には，大山らの協力の下，Cuckoo Sandbox v2.0.7 に ExMod の再現実装を施したものを使用した．デバッガ検知を行うプログラムは，解析環境のデバッガ検知回避性能の評価で広く用いられる Al-Khaser ベンチマークスイート [11] と文献 [12] から，例外処理を用いてデバッガ検知を行う計 3 種類のプログラムを用いた．また，実マルウェアにはマルウェアの大規模データベースである VirusShare[13] へ 2023 年にアップロードされた Windows マルウェアを 10,000 検体利用した．

3.1　デバッガの誤検知

Windows マルウェアの開発者は以下の例外処理機構を用いた例外ハンドラにデバッガ検知処理を定義する．

(1) ベクトル化例外処理（Vectored Exception Handling: VEH）

(2) 構造化例外処理（Structured Exception Handling: SEH）
(3) トップレベル例外フィルタ（Top-level Exception Filter）

各機構を使用する例外ハンドラを VEH ハンドラ，SEH ハンドラ，トップレベル例外ハンドラと呼ぶ．OS による例外ハンドラの探索は，上記の機構の順で行われる．VEH ハンドラは，任意のタイミングで登録と解除が可能であり，例外発生地点に依らず当該ハンドラが優先的に例外処理を行う．SEH ハンドラは，__try ブロックに対応する__except ブロックを指す．__try ブロックと対応する__except ブロックを合わせて try-except 文と呼ぶ．なお，try-except 文はネストが可能である．ネストしている場合，内側の__except ブロックから順に例外処理が試行される．マルウェアはデバッガ検知を目的として実行中に SEH ハンドラの登録を行う場合がある．OS は実行中のアプリケーションの SEH ハンドラを連結リスト構造で管理しており，FS レジスタが先頭レコードを指すアドレスをもつ．先述のネスト構造は連結リスト構造に対応しており，先頭レコードに対応する SEH ハンドラから順に実行される．マルウェアは FS レジスタの値を自身の SEH ハンドラの先頭アドレスに書き換えた後に例外を発生させ，自身の SEH ハンドラの実行の有無によりデバッガ検知を行う．トップレベル例外ハンドラは任意のタイミングで登録可能であり，VEH や SEH で処理されない例外の処理を行う．VEH や SEH と同様に，デバッガがアタッチしている場合は実行されないため，マルウェアによるデバッガ検知に利用される．

3.1.1 VEH ハンドラを用いたデバッガ検知のトリガー

評価には，Al-Khaser[11] に含まれる Interrupt_3 を用いた．グローバル変数 SwallowedException の値が TRUE で初期化されており，VEH ハンドラが実行されると FALSE に書き換わる．デバッガをアタッチした状態で実行すると VEH ハンドラが実行されないため，変数の値が TRUE から変化しない．プログラムは VEH ハンドラの登録後，実行するとブレークポイント例外が発生する__debugbreak 関数を呼び出して意図的に例外を発生させる．その後，変数の更新の有無によりデバッガ検知を行う．

このプログラムを ExMod で実行すると当該変数の値が FALSE に変わる．これは，ブレークポイント例外がシャドウ例外ハンドラで対応する種類の例外ではないため，例外処理が本来の RtlDispatchException の処理に移譲され，VEH ハンドラの探索と実行が行われたからである．

そこで，シャドウ例外ハンドラで対応する例外が生じるようにプログラムを改変して再度評価したところ，変数の値は TRUE から変わらなかった．つまり，シャドウ例外ハンドラが実行される例外の場合には，VEH ハンドラの実行が阻害されるために変数が更新されず，評価用プログラムがデバッガを誤検知することを示している．

3.1.2 SEH ハンドラを用いたデバッガ検知のトリガー

評価には，文献 [12] に含まれる hardwarebreakpointsSEH を用いた．FS レジスタの書き換えによる SEH ハンドラの登録後，例外を発生させて登録した SEH ハンドラの実行を試みる．SEH ハンドラが実行されると，デバッガがアタッチしていない旨を示すポップアップメッセージが表示される．しかし，デバッガをアタッチした状態で実行すると SEH ハンドラが実行されないため，ポップアップメッセージは表示されない．

このプログラムを ExMod で実行するとポップアップメッセージは表示されない．つまり，シャドウ例外ハンドラの実行により SEH ハンドラの実行が阻害され，評価用プログラムがデバッガを誤検知したことを示している．

3.1.3 トップレベル例外ハンドラを用いたデバッガ検知のトリガー

評価には，Al-Khaser[11] に含まれる UnhandledExceptionFilter_Handler を用いた．グローバル変数 bIsBeinDbg の値が TRUE で初期化されており，トップレベル例外ハンドラが実行されると FALSE に書き換わる．デバッガがアタッチした状態で実行すると，トップレベル例外ハンドラが実行されず，変数の値が TRUE から変化しない．プログラムは，トップレベル例外ハンドラの登録後，例外を発生させる Windows API である RaiseException 関数を呼び出して例外を発生させる．その後，変数が更新されるか否かを確認してデバッガ検知を行う．

このプログラムを ExMod で実行すると，当該変数の値が FALSE に変わる．これは，RaiseException の呼び出しによって生じる例外の処理を，ExMod は本来の RtlDispatchException の処理に移譲するからである．

そこで，RaiseException を用いずに例外が生じるようにプログラムの改変を行い，再度評価を行ったところ，変数の値が TRUE から変わらなかった．つまり，3.1.1 項と同様にシャドウ例外ハンドラが実行される場合には，トップレベル例外ハンドラの実行が阻害され，評価用プログラムがデバッガを誤検知することを示している．

3.2 不要なループの誤検知

ExMod において，不要なループからの脱出が記録されたマルウェアについて，当該ループ部のコードを調査した．この結果，ループ部と関係のない地点での例外発生時にも不要なループを実行中であると判断し，ExMod が脱出操作を行うケースが存在した．これは，不要なループの実行中であるか否かを判断する基準に問題がある．

図 2 に問題の概要を示す．図 2 はループ内で，ExMod が不要なループを実行していると判断するしきい値回数だけ例外が繰り返し発生する（図 2 の処理 1）．そして，ExMod が脱出操作を行い，当該ループから抜けた先の地点（図 2 の処理 2）でも，例外が生じるという流れである．

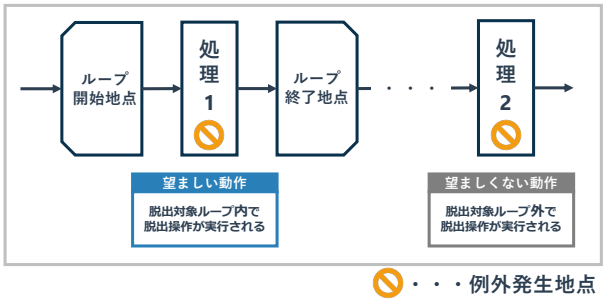

図 2 不要なループの誤検知
Fig. 2 False Detection of Unnecessary Loop.

この時，例外が発生した地点は一意に 2 ヵ所である．しかし，ExMod は，発生箇所に無関係に例外の発生回数を数えて，しきい値回数以上の例外発生と判断し，再び脱出操作を行う．このような脱出すべきループと関係ない地点での脱出操作の実行は，当該地点から先の処理の実行を阻害する恐れがあり，望ましくない．

3.3 不要なループからの脱出操作の成否

実際のマルウェアで観測された，不要なループについて説明する．また，その実行時に行われた ExMod の脱出操作とその結果について述べる．

3.3.1 ループカウントにレジスタを用いる不要なループ

図 3 のコードは，EAX レジスタをループカウンタに用いたループである．EAX レジスタの値を 4 行目でデクリメントし，当該レジスタの値が 0 になるまでループを繰り返す．2 行目の MOV 命令を実行すると例外が発生するため，ループの反復ごとにこの地点で例外が発生する．

このループを含む検体の解析結果から，しきい値回数だけループが実行された後，ループの反復ごとに 2 行目で脱出操作が行われたものの，失敗を続けたことが分かった．この後のループの反復で 2 行目の MOV 命令の実行により再び例外が発生すると，ExMod は実行中の関数からのリターンを行ったため，呼び出し元から実行を再開した．実行継続したものの，ループの直後の処理から再開できていないため，不完全な脱出である．

また，ループカウンタに ECX レジスタを用いた当該ループの亜種を含む別の検体でも，脱出操作の実行が確認された．こちらは，しきい値回数分ループした後，ループの反復ごとに脱出操作が失敗した．そして，再びループが反復し，2 行目の命令の実行により例外が発生すると，ExMod は ECX レジスタの値を 0 に書き換えた．その後，4 行目の DEC 命令を実行すると ECX レジスタの値がデクリメントされて値がアンダーフローしたため，5 行目の JNZ 命令がループからの脱出を行うための条件が満たされず，ループの実行が続いた．そして，ループの反復ごとに脱出操作は失敗を続け，最後は OS が実行を終了させた．これらのループから脱出させるためには，5 行目の JNZ 命令の直後の命令から実行を再開させる必要がある．

```
1  loop_start:
2  MOV           byte ptr [ECX],0x0
3  INC           ECX
4  DEC           EAX
5  JNZ           loop_start
```

図 3 ループカウントにレジスタを用いる不要なループ
Fig. 3 Unnecessary Loop Using Register for Loop Counting.

```
1  loop_start:
2  MOV           ECX,dword ptr [ECX]
3  CMP           ECX,-0x1
4  JZ            label_somewhere
5  CMP           dword ptr [ECX],EAX
6  JNZ           loop_start
```

図 4 ゼロフラグが立つまで繰り返す不要なループ
Fig. 4 Unnecessary Loop that Repeats until Zero Flag Is Set.

```
1  loop_start:
2  MOV           CL,byte ptr [EAX]
3  TEST          CL,CL
4  JZ            loop_end
5  ADD           EAX,dword ptr [EBP + 0x10]
6  JMP           loop_start
7  loop_end:
8  MOV           EDX,ECX
```

図 5 条件変数にアクセスできない不要なループ
Fig. 5 Unnecessary Loop with Inaccessible Condition Variable.

3.3.2 ゼロフラグが立つまで繰り返す不要なループ

図 4 のコードを含むマルウェアでは，2 行目と 5 行目の命令を実行すると例外が発生した．5 行目の CMP 命令が正常に実行されないため，ループから脱出する条件が満たされず，6 行目の JNZ 命令を実行するとループの先頭にジャンプする．しかし，しきい値回数だけループが実行され，一度目の脱出操作が失敗した次の反復で，5 行目の命令実行後，ExMod は EFLAGS レジスタのビット反転を行った．ゼロフラグの反転により 6 行目の JNZ 命令がループを脱する条件が満たされ，ループの直後の命令から実行を再開した．ループから脱出させるためには，図 3 の場合と同様に，ループの先頭にジャンプする命令の直後の命令から実行を再開させる必要があると考えられる．

3.3.3 条件変数にアクセスできない不要なループ

図 5 のコードを含むマルウェアは，2 行目の命令を実行すると例外が発生した．このため，3 行目の TEST 命令のオペランドである CL レジスタにループからの脱出に必要な値が設定されないため，4 行目において JZ 命令を実行してもループから脱出せず，6 行目の JMP 命令でループの先頭へのジャンプを繰り返した．しきい値回数だけループ

を実行した後，ループの反復ごとに 2 行目で行われた脱出操作は失敗を続け，最後は OS が実行を終了させた．ループから脱出させるためには，8 行目の MOV 命令から実行を再開させる必要があると考えられる．

3.4 評価に基づく要件

評価を踏まえ，例外発生マルウェアの実行継続手法は，以下の 2 つの要件を満たす必要があると考える．

- 継続処理がデバッガの誤検知を引き起こさないこと
- ループの直後から実行再開して不要なループから脱出させること

特に，不要なループからの脱出操作は当該ループ区間内でのみ実行されるべきである．

4. 提案手法

本章では，3.4 節の要件を踏まえた提案手法について述べる．**図 6** に全体像を示す．例外が発生すると，OS が RtlDispatchException を呼び出し，発生例外を処理可能な例外ハンドラの探索を行う（図 6 の Step 1）．該当する例外ハンドラが存在する場合，当該例外ハンドラを実行する（図 6 の Step 1-a）．しかし，発生例外を処理可能な例外ハンドラが存在しない場合，あるいは実行したトップレベル例外ハンドラの返り値が OS にアプリケーションの終了を指示する値である場合，OS が NtTerminateProcess を呼び出してアプリケーションを終了させる．提案手法は NtTerminateProcess によるアプリケーションの終了を防ぎ（図 6 の Step 1-b），実行を継続させる．不要なループの実行中ではないと判断する（図 6 の Step 2）と，後続処理から実行を再開させる（図 6 の Step 2-a）．不要なループの実行中であると判断する（図 6 の Step 2）と脱出操作を行い，不要なループから脱出させる（図 6 の Step 2-b）．

提案手法は，例外ハンドラの探索と実行の過程に影響を一切与えない．このため，例外ハンドラの実行の有無で判断する，いかなるデバッガ検知をも誤ってトリガーしない．

4.1 デバッガ検知を回避した後続処理からの実行再開

4.1.1 実行再開方針

実行再開方針の検討に際し，評価検体で生じた例外の種類を調査したところ，スタックやヒープの崩壊あるいは，高深度の再帰呼び出しによるオーバーフローにより例外が生じる場合があると分かった．この場合，新たに関数を呼び出せないなどの理由から，実行継続が困難であると考えられる．また，例外は計 21 種類発生しており，対応を一部の例外に限定する再開処理では対応が不十分である．

そこで，スタックやヒープのオーバーフローを示す例外が発生した場合は実行中の関数からのリターンを行い，崩壊を示す例外が生じた場合はマルウェアの実行を終える．そして，これら以外の例外が生じる場合は，例外発生命令

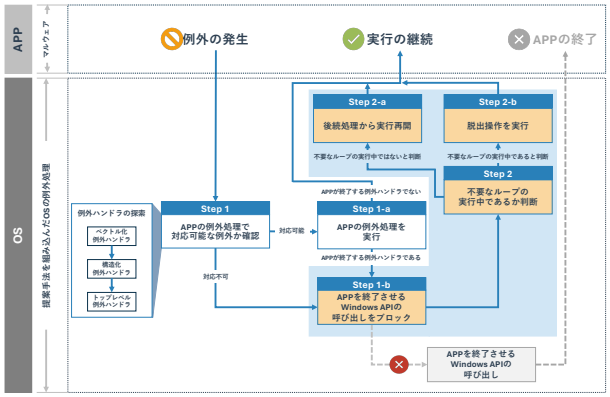

図 6 提案手法の全体像
Fig. 6 Overview of Proposal.

の次の命令から実行を再開させることを基本方針とする．

4.1.2 実行再開処理

Windows では例外処理がスレッドごとに行われる．RtlDispatchException は，例外が発生したスレッドのコンテキスト情報を引数にもつ．コンテキスト情報とは，スレッドの実行状態を表すレジスタ，プログラムカウンタ，スタックポインタを指す．はじめに，提案手法では例外が生じて RtlDispatchException が呼び出されると，引数のコンテキスト情報をスレッドローカルストレージに保存する．スレッドローカルストレージとは，スレッドごとのデータ保存領域を指す．

発生例外を処理できる例外ハンドラが見つからない場合

この場合，NtTerminateProcess の呼び出しは RtlDispatchException のフックコードの改変により防止できる．そこで，VEH ハンドラや SEH ハンドラで発生例外を処理できず，トップレベル例外ハンドラが登録されていない場合，オリジナルの RtlDispatchException の処理から，フックコードへ処理を移譲させる．そして，処理が移譲されたフックコード内で，4.1.1 項の方針に基づく実行再開処理を行う．具体的には，実行中の関数からのリターンを行う場合は，呼び出し元のスタックフレームを指すよう，コンテキスト情報内のスタックポインタの値を修正する．加えて，呼び出し元へのリターンアドレスをコンテキスト情報内のプログラムカウンタへ書き込む．また，例外発生命令の次の命令から再開させる場合，ディスアセンブラエンジンの Capstone[14] を用いて例外発生命令の長さを算出し，コンテキスト情報内のプログラムカウンタへ加算する．長さを算出できない不正な命令の場合，プログラムカウンタのインクリメントを行う．最後に，コンテキスト情報が指す実行状態からスレッドの実行を再開できる Windows API である NtContinue を呼び出し，実行を再開させる．

実行したトップレベル例外ハンドラの返り値が OS に実行終了を指示する値である場合

この場合，NtTerminateProcess の呼び出しを未然に防止できない．しかし，引数に渡されるステータスコードか

ら，例外処理に起因した呼び出しであるか否かを判定できる．そこで，NtTerminateProcess へのフックコード中でステータスコードを確認し，例外処理に起因した呼び出しである場合は実行再開処理を行い，そうでない場合は本来の処理に移譲し，アプリケーションを終了させる．実行再開処理の内容は，発生例外を処理できる例外ハンドラが見つからない場合と同様である．しかし，NtTerminateProcess は例外発生時のコンテキスト情報を引数にもたないため，スレッドローカルストレージに保存したコンテキスト情報を用いて実行を再開させる．

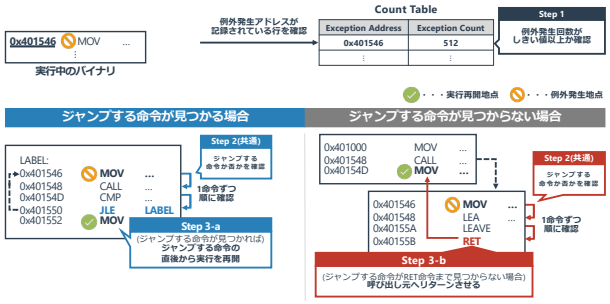

図 7　不要なループの検知および脱出操作
Fig. 7　Detection and Escape from Unnecessary Loop.

4.2　不要なループからの脱出

提案手法が行う不要なループの検知と当該ループからの脱出操作を**図 7** に示す．

4.2.1　不要なループの検知

実行中に例外が発生した命令のアドレスと，その地点で例外が発生した回数をカウント表（図 7 の Count Table）に記録する．以下，例外が発生した命令のアドレスを例外発生アドレスと呼ぶ．まず，例外発生アドレスがカウント表に記録されているか，例外が生じるたびに確認する．例外発生アドレスが過去に一度も記録されていない場合，例外発生アドレスと例外発生回数をカウント表に新たに記録する．過去に記録済みのアドレスである場合，対応する行の例外発生回数をインクリメントする．そして，不要なループの実行中であるか否かを，4.1 節の実行再開処理の前にカウント表を用いて判断する．例外発生アドレスに対応する例外発生回数の記録を参照し，しきい値回数以上の例外発生を確認した（図 7 の Step 1）場合は実行再開処理ではなく，不要なループからの脱出操作を行う．不要なループの実行中と判断する例外発生回数は，大山らの研究に基づき 512 回とした．

4.2.2　不要なループからの脱出操作

脱出操作は 3 章での検討結果より，ループの先頭にジャンプする命令の直後から実行を再開させることが望ましいと判断した．そこで，提案手法はこれを実現するため，例外発生命令からアドレス高位に向かって順に命令を辿り，ジャンプする命令が存在するか否かを確認する（図 7 の Step 2）．x86 命令は処理内容が同一であっても，命令の種類を表す数値であるオペコードの値が異なる場合がある [15]．そこで，Capstone を用いて命令の名前を表す文字列であるニーモニックを取得し，ニーモニックがジャンプする命令のものであるか否かの判定を行う．処理内容が同一であればニーモニックは同一の文字列となる．

ジャンプする命令が見つかる場合

ジャンプする命令が存在するアドレス値に，ジャンプする命令の長さを加算した値を例外発生時のコンテキスト情報のプログラムカウンタに書き込む．そして，4.1.2 項と同様の手順でジャンプする命令の直後の命令から実行再開を行い，ループから脱出させる（図 7 の Step 3-a）．しかし，しきい値回数目でジャンプする命令の直後から再開させたにも関わらず，再び同じ地点で例外が発生した場合は，この脱出操作が有効ではないと考えられるため，4.1.2 項と同様の手順で呼び出し元へリターンさせる．

ジャンプする命令が見つからない場合

呼び出し元へリターンする命令が見つかるまでにジャンプ命令が見つからない検体が存在する．例外が生じるコードを含む関数を繰り返し呼び出すため，カウント表からは一見，不要なループが存在するように見えるのである．そこで，このような場合は 4.1.2 項と同様の手順で呼び出し元へリターンさせる（図 7 の Step 3-b）．

5.　評価

提案手法を Cuckoo Sandbox v2.0.7 に実装し，3 章と同様の環境を用いて評価を行った．なお，不要なループからの脱出性能の評価には，図 3，図 4 と図 5 のループを C 言語のインラインアセンブリ機能を用いて再現したサンプルプログラムを使用した．

5.1　実行再開処理のデバッガ検知回避性能

評価結果を**表 2** に示す．表では，3 章で示した例外ハンドラを用いたデバッガ検知プログラムの実行結果を ExMod と比較している．ExMod の評価において，例外の発生方法によってはデバッガの誤検知を引き起こしたプログラムにおいても，提案手法はデバッガの誤検知を引き起こさなかった．

5.2　不要なループからの脱出性能

評価結果を**表 3** に示す．表では図 3，図 4 と図 5 のループを再現したサンプルプログラムの実行結果を ExMod と比較している．図 3 と図 4 のループでは，例外発生地点でしきい値回数分の例外が発生すると提案手法によりループの先頭に戻るジャンプ命令の直後から実行を再開して脱出に成功している．ところが，図 5 のループからは脱出できない場合があった．例外発生命令からアドレス高位に向かって命令を辿るために命令長を取得する Capstone の処

表 2　デバッガ検知回避性能
Table 2　Performance of Avoiding Debugger Detections.

	提案手法	ExMod
VEH ハンドラ	○	△
SEH ハンドラ	○	×
トップレベル例外ハンドラ	○	△

○: デバッガ検知を引き起こさない.
△: 特定の場合にデバッガ検知を引き起こさない.
×: デバッガ検知を引き起こす.

理で例外が生じる場合があるからである．この例外を処理すべく OS が行う例外処理の過程においても提案手法による命令長の取得処理で例外が生じるため，再帰的に例外の発生が続く．これは，本研究で用いるバージョンの Capstone には問題のあるデコード処理を取り止める機能が存在しないからである．当該機能をもつ現行の Capstone の使用により解決可能な問題であると考えられるため，図 5 のループにおける提案手法の評価は △（制限有）としている．

6. まとめ

本論文では，動的解析において，例外発生に伴うマルウェアの実行および解析が中断するという問題を指摘し，その問題を解決するために ExMod の手法に注目していることを述べた．その後，ベンチマークプログラムや実際のマルウェアを用いた ExMod の評価結果を示し，例外発生マルウェアの実行継続を実現する上で求められる要件を明らかにした．具体的には，デバッガ検知を誤ってトリガーしない実行再開処理の実現が必要なこと，および，不要なループからの脱出はループの先頭にジャンプする命令の直後から実行を再開させる必要があることの 2 つを示した．そして，提案手法は，例外ハンドラの探索と実行を阻害しない実行再開処理と，ジャンプする命令の直後の命令から実行を再開させる脱出操作により，これらの要件を満たすことを示した．

提案手法の有効性を検証するため，7 つのシンプルなプログラムを用いて評価を行った．評価の結果，デバッガ検知回避性能の評価において，ExMod では誤検知を引き起こしたすべてのプログラムで誤検知を引き起こさないことを示した．また，不要なループからの脱出性能評価においては，ExMod と比較して脱出性能が向上したものの，一部のループからの脱出に課題が残ることを示した．

今後は脱出できなかったループからの脱出を実現し，手法の有効性を向上させる．

謝辞　ExMod の再現実装にあたり，多くの御助言を頂きました筑波大学の大山恵弘氏に深く御礼申し上げます．

表 3　不要なループからの脱出性能
Table 3　Performance of Escaping from Unnecessary Loops.

	提案手法	ExMod
図 3 のループ（EAX レジスタを使用）	○	△
図 3 のループ（ECX レジスタを使用）	○	×
図 4 のループ	○	○
図 5 のループ	△（制限有）	×

○: ループ直後の処理から実行を再開する.
△: ループから脱するが，呼び出し元からの再開である.
×: ループから脱出できない.

参考文献

[1] 情報処理推進機構 (IPA)：情報セキュリティ 10 大脅威 2024, https://www.ipa.go.jp/security/10threats/10threats2024.html (Accessed 2024-08-06).
[2] Ball, T.: The concept of dynamic analysis, *ACM SIGSOFT Software Engineering Notes*, Vol. 24, No. 6, pp. 216–234 (1999).
[3] 吉川孝志：標的型攻撃ランサムウェア「Ryuk」の内部構造を紐解く, https://www.mbsd.jp/research/20191211/ryuk/ (Accessed 2024-08-06).
[4] Peng, F., Deng, Z., Zhang, X., Xu, D., Lin, Z. and Su, Z.: {X-Force}:{Force-Executing} binary programs for security applications, *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 829–844 (2014).
[5] You, W., Zhang, Z., Kwon, Y., Aafer, Y., Peng, F., Shi, Y., Harmon, C. and Zhang, X.: Pmp: Cost-effective forced execution with probabilistic memory preplanning, *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, pp. 1121–1138 (2020).
[6] Oyama, Y. and Kokubo, H.: Forced continuation of malware execution beyond exceptions, *Journal of Computer Virology and Hacking Techniques*, Vol. 19, No. 4, pp. 483–501 (2023).
[7] 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*, Vol. 1, No. 2012, pp. 1–27 (2012).
[8] Andrea Allievi, Alex Ionescu, Mark E. Russinovich, David A. Solomon: インサイド Windows 第 7 版 下, 株式会社日経 BP (2022).
[9] Or-Meir, O., Nissim, N., Elovici, Y. and Rokach, L.: Dynamic malware analysis in the modern era—A state of the art survey, *ACM Computing Surveys (CSUR)*, Vol. 52, No. 5, pp. 1–48 (2019).
[10] Bremer, J.: Blackhat 2013 workshop: Cuckoo sandbox-open source automated malware analysis (2013).
[11] LordNoteworthy: Al-Khaser, https://github.com/LordNoteworthy/al-khaser/ (Accessed 2024-08-06).
[12] Tamiollo, D.: Anti-Debug-examples-Windows, https://github.com/domin568/Anti-Debug-examples-Windows (Accessed 2024-08-06).
[13] Corvus Forensics: VirusShare.com - Because Sharing is Caring, https://virusshare.com/ (Accessed 2024-08-06).
[14] Quynh, N. A.: Capstone: Next-gen disassembly framework, *Black Hat USA*, Vol. 5, No. 2, pp. 3–8 (2014).
[15] Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual (2024).