Computer Security Symposium 2019
21 - 24 October 2019

例外を発生させるマルウェアのための動的解析手法

大山 恵弘[1]　小久保 博崇[2]

概要：マルウェアは例外によって実行を終了することがある．それらの例外の中には，実行条件が変わると発生しなくなるものも存在する．マルウェアの潜在的脅威を理解するには，例外が発生しない場合に観測される挙動を把握する必要がある．しかし，既存の多くの解析システムはそのための仕組みを有していない．本研究では例外を発生させるマルウェアの動的解析を支援する手法を提案する．その手法では，実行終了をもたらす例外をマルウェアが発生させたら，そのマルウェアが終了しないようにメモリやレジスタを書き換えて実行を継続させる．その手法により，例外が発生しない条件が不明でも，その条件下で観測される挙動を予測できるようになる．著者らはその手法に基づくシステムを開発してマルウェアの動的解析を行った．

キーワード：例外，マルウェア，動的解析，サンドボックス

Dynamic Analysis Method for Exception Raising Malware

Yoshihiro Oyama[1]　Hirotaka Kokubo[2]

Abstract: Malware programs sometimes terminate the execution because of exceptions. Some of these exceptions do not occur under different execution conditions. To understand the potential threat of malware, analysts need to collect the information on the malware behavior observed when these exceptions do not occur. However, many existing analysis systems do not have a mechanism for it. In this study, we propose a technique to support the dynamic analysis of malware programs that raise exceptions. If an analyzed malware program raises an exception that causes execution termination, the technique modifies the memory and registers of the malware to prevent it from being terminated, and then lets it continue the execution. The technique enables analysts to predict the behavior observed when the exception is not raised, even if they have not identified the condition under which the exception is not raised. The authors developed a system based on the technique and conducted dynamic analysis of malware using it.

Keywords: Exceptions, malware, dynamic analysis, sandboxes

1. はじめに

マルウェアの挙動や目的を理解するために，マルウェアを実行せずにコードを人やツールが調べる静的解析と呼ばれる解析と，マルウェアをサンドボックスやデバッガの上で実行する動的解析と呼ばれる解析が用いられている．動的解析には，静的解析では計算量や時間の面で難しい挙動も観察することができるという利点がある．

動的解析では，マルウェアが例外を発生させ直後に実行を終了することがしばしば起きる．例外が発生する理由としては様々なものが考えられるが，一部の例外は，そのマルウェアが動作している実行環境の構成や設定，外部からの入力などの，マルウェアの作者が想定しなかった条件によるものである可能性がある．そのようなマルウェアは，異なる条件下では例外を発生させずに実行を続ける可能性がある．一般に，マルウェア解析では，たとえ特定の条件下でのみ観測される挙動であっても，潜在的に実行されるすべての挙動を明らかにすることが望まれる．解析対象のマルウェアが実行するかもしれない処理を知ることは，潜

[1] 筑波大学
University of Tsukuba
[2] 株式会社富士通研究所
FUJITSU LABORATORIES LTD.

在的脅威の認識に有用という観点から，たとえその実行条件がわからなくても意味があることがある．

マルウェアの動的解析に用いられる既存のサンドボックスでは通常，例外に関しては，発生した例外の概要を記録して解析者に提示する程度の受動的な処理だけを実行する．以降の実行に対して影響を及ぼすような能動的な処理は実行しない．例えば，大半のサンドボックスは例外の発生によりマルウェアが実行を終了する場合でも，例外の発生や実行の終了を妨げない．その結果，もしその例外が発生しなかったり例外への対処方法が違っていたら，先の実行はどうなっていたかを解析者が予想する役には立たない．その例外を発生させない条件を見出すには，多様な実行環境を作成したり，多様な入力を与えるなどの対策が必要になるが，それには多大な人的コストや時間コストがかかる．

そこで本研究では，マルウェアがサンドボックス上で発生させる例外を分析するとともに，例外への対処に関してプログラムの外部から能動的に影響を及ぼすことにより動的解析を支援する手法を提案する．対象とするマルウェアは Windows 向けの PE 形式のバイナリプログラムとする．

まず本論文では，マルウェアがどのような例外をどの程度発生させるかについて調査した結果を述べる．その調査では，FFRI Dataset 2017 [8] に含まれる動的解析の結果と，著者らが別のマルウェアに対して動的解析および逆アセンブルを行った結果を用いた．次に，発生した例外の効果を無効化または改変してからマルウェアに実行を継続させる動的解析手法を提案する．その手法ではマルウェアを実行するサンドボックスが例外発生時のマルウェアのメモリとレジスタを検査し，例外の種類に応じて，実行を継続させるための適切な修正を施す．例えば，マルウェアが無効なメモリアドレスから値を読もうとして例外を発生させた場合には，その例外が発生しなかったかのようにマルウェアプロセスの状態を修正し，例外を発生させた命令をエミュレーション実行する．エミュレーション実行では，サンドボックスが決める 0 などの値を読み出したことにして，その値を書き込み先に格納する．

厳密には，このサンドボックスは解析対象プログラムのセマンティクスを改変しており，元のプログラムとは別のプログラムを解析しているとみなせる．本研究では，マルウェアの実行を元の実行から少々変更することは許容し，厳密に正確な情報が得られなくても，ある程度正しくて挙動の予測に役立つ情報が得られれば良いという立場に立つ．マルウェアの潜在的挙動の情報が得られることは，たとえその情報の信頼性が高くなくても，得られないことよりはるかに良いという考え方が本研究の背景にある．

我々は上記の手法を実現するシステム IgnorEx を，Cuckoo Sandbox（以降では Cuckoo と表記）の拡張によって実装した．さらに，IgnorEx 上でマルウェアの動的解析を行った．その解析では，元々は例外によって途中で終了していたマルウェアの実行の多くを継続させることができ，元の Cuckoo 上での実行では現れなかった多くの挙動を引き出すことができた．

```
__exception__(stacktrace=["SetFunctionAddresses+0x2a6d1 @
0x2d9fad1", ...], exception={"instruction_r"=>"8b 0a bf ff
fe fe 7e 8b c1 8b f7 33 cb 03 f0 03", "instruction"=>"mov
ecx, dword ptr [edx]", "exception_code"=>"0xc0000005",
"symbol"=>"SetFunctionAddresses+0x39e58",
"address"=>"0x2daf258"}, registers={"esp"=>2290736,
"edi"=>4, "eax"=>14906, "ebp"=>2290768, "edx"=>0,
"ebx"=>976894522, "esi"=>2906016672, "ecx"=>2290840})

__exception__(stacktrace=["EbGetHandleOfExecutingProject+0x22b3
...", ...], exception={"instruction_r"=>"c9 c2 10 00 89 45 c0
eb ed 64 a1 18 00 00 00 8b", "symbol"=>"RaiseException+0x54
...", "instruction"=>"leave ", "module"=>"KERNELBASE.dll",
"exception_code"=>"0xc000008f", "offset"=>46944,
"address"=>"0x75b9b760"}, registers={"esp"=>1241368,
"edi"=>2444792, "eax"=>1241368, "ebp"=>1241448, "edx"=>0,
"ebx"=>2444792, "esi"=>2444792, "ecx"=>2})
```

図 1　データセットに含まれる例外情報の例

2. 例外

2.1 Windows における例外

Windows 上での C++や C#などのいくつかの言語のプログラムでは，不正なメモリ領域へのアクセスやゼロ除算などの例外的事象の抽象化である例外を用いることができる．例外には種類ごとに例外コードが付けられている．

例外発生時にはプログラムは例外の種類ごとに定められた処理（例外ハンドラ）を実行できる．例外ハンドラでは実行を終了することもできれば，変数の値の更新などを行った後に実行を続けることもできる．発生した例外に対するハンドラをプログラムが指定していない場合も多いが，その場合には Windows や言語処理系が定めたハンドラが実行される．例えば，不正なメモリアクセスを行うと例外が発生するが，その例外に対するハンドラが指定されていなければ，実行を終了する処理が行われる．

例外は不正なメモリアクセスなどの例外的操作を契機に OS が発生させることもあり，ユーザプログラムが API 関数を通じて発生させることもある．

2.2 FFRI Dataset 2017 に記録された例外発生

FFRI Dataset 2017 はマルウェア 6251 検体を Cuckoo 上で実行した動的解析結果のデータセットである．FFRI Dataset 2017 にはマルウェアの実行中に発生した例外の情報も記録されている．データセットに含まれる API コール列の情報の中に，__exception__という名前の API 関数が呼び出されたかのような形式で例外の情報が埋め込まれている．その情報には，例外コード，例外の原因となったメモリアドレス，例外発生時のスタックトレース，プログラムカウンタ以降の数バイト分のバイト列とその逆アセンブル結果，汎用レジスタの値などが含まれる．そのデータセットは JSON 形式であるが，例外の情報の部分を人が読みやすいように整形した出力の例を**図 1** に示す．

表 1　FFRI Dataset 2017 に記録された実行における例外発生回数

	例外コード		発生回数	検体数	マルウェア名数
1	c0000005	(STATUS_ACCESS_VIOLATION)	9808	887	125
2	c000008f	(STATUS_FLOAT_INEXACT_RESULT)	9388	296	44
3	c0000096	(STATUS_PRIVILEGED_INSTRUCTION)	5236	376	38
4	e0434f4d	(Exception related to SQL Server or .NET CLR)	4038	313	60
5	c000001d	(STATUS_ILLEGAL_INSTRUCTION)	3032	377	37
6	c0000094	(STATUS_INTEGER_DIVIDE_BY_ZERO)	1491	59	19
7	c0000002	(STATUS_NOT_IMPLEMENTED)	1240	1240	18
8	80000003	(STATUS_BREAKPOINT)	913	35	17
9	c000001e	(STATUS_INVALID_LOCK_SEQUENCE)	679	7	4
10	c00000fd	(STATUS_STACK_OVERFLOW)	588	109	8
11	0eedfade	(Delphi-related exception)	584	183	21
12	80000004	(STATUS_SINGLE_STEP)	51	14	6
13	406d1388	(Exception related to Visual C++ or threads)	36	20	12
14	800706ba	(Exception related to availability of RPC servers)	22	22	3
15	4001000a	(DBG_PRINTEXCEPTION_WIDE_C)	9	8	3
16	e0434f4e	(EXCEPTION_HIJACK)	2	2	2
17	c000071c	(STATUS_INVALID_THREAD)	1	1	1
18	c06d007e	(Exception related to license or module lookup)	1	1	1
19	8001010d	(RPC_E_CANTCALLOUT_ININPUTSYNCCALL)	1	1	1
総数			37120	3951	211

FFRI Dataset 2017 に記録された実行における，例外が発生した回数や例外を発生させた検体の数を**表 1** に示す．表中の括弧内の大文字のみで書かれた文字列は標準的な C++プログラムで各例外コードに対応するシンボルである．以降の文章では，これらの文字列から（もしあれば）STATUS_ を削除した文字列によって各種類の例外を表す．対応するシンボルが見当たらない場合や複数見つかった場合には，著者らによるその例外の説明が書かれている．「発生回数」の列は各例外が発生した回数を示している．「検体数」の列は各例外が 1 回以上発生した検体の数を示し，「マルウェア名数」の列はそれらの検体のマルウェア名の種類数を示している．マルウェア名としてはデータセットに記録された Microsoft 製品による判定結果を用いた．Microsoft 製品がマルウェアと判定していない検体はすべて UNKNOWN という同一のマルウェア名であるとした．「総数」の行に書かれた数はそれぞれ例外発生の総数，1 回以上例外が発生した検体の数（その列の上に書かれた数の総和，すなわち延べの数とは異なる），それらの検体のマルウェア名の種類数を示している．1 回以上例外を発生させた検体の数は 3951 である．全検体の数は 6251 であるので，それは全検体数の約 63%である．例外発生の総数は 37120 であり，1 検体あたり平均約 6 回例外が発生している．1 回以上例外を発生させた検体に限ると，1 検体あたり平均約 9.4 回例外が発生している．発生回数の点でも発生した検体のマルウェア名の種類数の点でも，最多は ACCESS_VIOLATION 例外である．発生回数では次が僅差で FLOAT_INEXACT_RESULT 例外である．発生させた検体の数が最も多い例外は NOT_IMPLEMENTED であり，それらの検体は 1 回ずつこの例外を発生させる．

2.3　個々の例外の詳細

表 1 の上位 10 件の例外を取り上げて説明する．

2.3.1　ACCESS_VIOLATION 例外

ACCESS_VIOLATION 例外は，アクセスできないメモリアドレスに対する読み出しまたは書き込みにより発生する．例えば図 1 の 1 つ目の例外情報は，EDX レジスタの値のアドレスすなわちアドレス 0 から読み出しを試みて例外が発生したことを示している．他にも多くの検体の実行において，様々なスタックやレジスタの状態で，様々な命令で，不正なメモリアドレスをアクセスして ACCESS_VIOLATION 例外が発生している．この例外に対して特に対処しなければ，プログラムは終了することになる．

2.3.2　FLOAT_INEXACT_RESULT 例外

FLOAT_INEXACT_RESULT 例外は，浮動小数点数演算の結果を小数で正確に表現できない場合に発生する．それには整数でない浮動小数点数を整数に変換する場合も含まれる．マルウェアの大部分の処理では浮動小数点数演算は不要と予想するが，マルウェアには浮動小数点数演算を実行するものもある．例えば，浮動小数点数の除算によって CPU 周波数を求めるマルウェアが存在する [4]．

Windows では浮動小数点数演算の例外は，ユーザプログラムが明示的に発生を有効化しない限り，ユーザプログラムには送られない．その結果，例外が発生するはずの演算を行っても，例外のハンドラは実行されない．一般に，浮動小数点数演算における例外がプログラムの実行の継続を困難にすることは稀である．実際，データセットに記録された大半の実行でも，浮動小数点数演算の例外の発生後にも実行は長く続いている．

2.3.3　PRIVILEGED_INSTRUCTION 例外

PRIVILEGED_INSTRUCTION 例外は，現在の CPU の特権レベルで実行が許可されていない命令の実行を試みると発生する．例えば，高い特権レベルを必要とする命令を低い特権レベルで実行しようとすると発生する．そのような命令は通常カーネルやデバイスドライバのコードに含ま

れており，例えば，入出力命令，割り込み制御命令，特殊レジスタにアクセスする命令である．この例外に対して特に対処しなければ，プログラムは終了する．

データセットに記録された実行では，376 検体が計 5236 回もの多数回，この例外を発生させている．一方，それらの例外を発生させた命令は 2 種類しかなく，IN 命令（ストリング拡張版も含む）と STI 命令である．IN 命令は I/O ポートからデータを読む特権命令であり，STI 命令は割り込みを許可する特権命令である．低い特権レベルでこれらの命令を実行すると例外が発生する．

例えばデータセット中の以下の例外情報は，IN 命令で例外が発生したことを示している．アクセスされている 22104（0x5658）番ポートは，VMware が仮想マシンとハイパバイザの間で情報をやり取りするために提供しているポートである．マルウェアはこのポートを通じて自身が VMware 上で動いているかどうかを知ることができる．

```
__exception__(stacktrace=["gstgnq+0xdbee @
0x40dbee", ...], exception={"instruction_r"=>"ed
81 fb 68 58 4d 56 0f 94 45 e7 5b 59 5a 83
4d", "symbol"=>..., "instruction"=>"in
eax, dx", "module"=>"gstgnq.exe",
"exception_code"=>"0xc0000096", "offset"=>54643,
"address"=>"0x40d573"}, registers={"esp"=>1244288,
"edi"=>2004590921, "eax"=>1447909480,
"ebp"=>1244348, "edx"=>22104, "ebx"=>0, ...})
```

PRIVILEGED_INSTRUCTION 例外を発生させる検体は，特権命令を実行することから，カーネルレベルマルウェアである可能性がある．また，STI 命令と IN 命令を含むいくつかの特権命令は 1 バイト命令であるため，命令列ではないデータを命令として実行した場合にも，低くない確率でそれらの命令は実行される．この例外が発生した場合には，それは正しい命令を実行した結果か，命令ではない部分を命令として実行した結果かを調べる必要がある．

2.3.4 SQL Server または.NET CLR に関する例外

例外コード 0xe0434f4d の例外についての Microsoft による公式文書は見つけていないが，この例外が SQL Server や.NET CLR に関連することを示す情報は存在する[*1,*2]．この例外はユーザレベルのアプリケーションやライブラリが意図的に発生させており，多くの場合にそれらが想定内の事象として例外を処理すると推測する．

2.3.5 ILLEGAL_INSTRUCTION 例外

ILLEGAL_INSTRUCTION 例外は無効な命令，すなわち，命令が割り当てられていないバイト列を実行したときに発生する．この例外に対して特に対処しなければ，プログラムは終了することになる．データセットに記録された実行におけるこの例外の発生の主要な理由は，命令列が存在しないメモリ領域にバグまたは作者の意図によってジャンプしたことであると推測している．例えば以下の例外情報によると 3e 3e 3e 3e ... というバイト列が命令として実行されているが，そのバイト列は明らかに命令列ではない．

```
__exception__(stacktrace=["048d390f....", ...],
exception={"instruction_r"=>"3e 3e 3e 3e 3e 3e
3e 3e 3e 3e 3e 3e 3e 3e 3e 3e", "symbol"=>"...",
"instruction"=>"ds ", "module"=>"...",
"exception_code"=>"0xc000001d", "offset"=>20948,
"address"=>"0x4051d4"}, registers={ ... })
```

2.3.6 INTEGER_DIVIDE_BY_ZERO 例外

INTEGER_DIVIDE_BY_ZERO 例外はゼロ除算によって発生する．具体的には，除数を与えるオペランドのレジスタやメモリに 0 が入っている状態で DIV 命令や IDIV 命令を実行すると発生する．マルウェアの多くはゼロ除算の例外を発生させた後も実行を終了せず，ファイル操作などの様々な操作を実行している．これは，その例外がユーザプログラムかライブラリのどこかの層で捕捉されて，さらに下の層には伝わらなかったからであると推測する．この例外にサンドボックスがどう対処すべきかは簡単ではない問題である．対処が必要ないと思われる場合も多いとともに，対処がマルウェアの挙動を大きく変える可能性もある．

2.3.7 NOT_IMPLEMENTED 例外

NOT_IMPLEMENTED 例外は，要求された操作が未実装であるときに発生する．データセットに記録された例外情報によると，この例外はすべて IWICColorContext_InitializeFromMemory_Proxy という関数から API 関数の RaiseException が呼び出されて発生している．この IWIC... という関数は windowscodecs.dll という DLL に含まれ，windowscodecs という関数から呼び出されている．この例外が発生する場所や条件はユーザプログラムの開発者が予測できることが多く，ユーザプログラムが例外を適切に処理して実行は継続することが多いと推測する．実際，この例外を発生させた検体の 99%以上が，例外発生後に 1000 以上の API コールを実行している．

2.3.8 BREAKPOINT 例外

BREAKPOINT 例外はブレークポイント（int3）命令の実行などの理由で発生する．マルウェアはこの例外を意図的に発生させることも，意図的ではなく発生させることもある．データセットには検体そのものは含まれず，int3 命令の周辺の命令列に関して得られる情報が限られるため，著者らが利用できるマルウェア検体を用いて説明する．ある検体は以下の命令列を含む．

```
403cc2: pop eax                 403ccd: add esp, 4
403cc3: xor eax, eax            403cd0: test eax, eax
403cc5: int3                    403cd2: je 0x40441f
403cc6: pop dword fs:[0]        403cd8: jmp 0x403cf1
```

この命令列は int3 命令によってデバッガを検出する典型的なコードである．このマルウェアはハンドラを定義した上でこのコードを実行してブレークポイント例外を発生さ

*1 https://support.microsoft.com/ja-jp/help/2742131/0xe0434f4d-exception-and-0xc001000a-error-message-when-you-create-a-ma

*2 https://social.msdn.microsoft.com/Forums/vstudio/en-US/af2252c5-bdba-4c1c-81ef-209c3f219246/clr-exception-code-e0434f4d?forum=clr

せ，ハンドラが実行されない場合には（その例外を捕捉する）デバッガが自身に張り付いていると判断する．

また，別のマルウェアは，意味があるとは思われない以下の命令列を実行した結果，この例外を発生させる．int3 命令は 0xcc という 1 バイトで表されるため，それが偶然実行される可能性は低くない．

```
d099: add BYTE PTR [eax],al    d0af: fist WORD PTR [eax-0x18]
d09b: mov DWORD PTR fs:0x0,esp d0b2: out dx,eax
d0a2: int3                     d0b3: (bad)
d0a3: fbld TBYTE PTR [eax+...] d0b4: (bad)
d0a7: add eax,0xffffffd4       d0b5: (bad)
d0ac: popf                     d0b6: fpatan
d0ad: jmp eax                  d0b8: stos DWORD PTR ...
```

2.3.9 INVALID_LOCK_SEQUENCE 例外

INVALID_LOCK_SEQUENCE 例外は無効なロックのシーケンスを実行しようとすると発生する．データセットに記録された実行では，この例外が発生するのは必ず以下の機械語コードと命令列においてである．

```
f0           invalid
f0           invalid
c7           invalid
c8 64 67 8f  enter 0x6764,-0x71
06           push es
00 00        add byte [eax],al
83 c4 04     add esp,4
```

0xf0 は lock-prefix に対応するコードである．また，それが 2 つ付加されているコード 0xc7 は，対応する命令が無い無効命令である．この命令列は正しいものであるとは思えないため，マルウェアは命令列ではない領域にジャンプしたなどの理由でこの命令列を実行したと推測する．この例外が発生した場合には，プログラムが異常な場所を実行している可能性を考慮する必要がある．

2.3.10 STACK_OVERFLOW 例外

STACK_OVERFLOW 例外はスタックオーバフローにより発生する．一般にこの例外が発生する理由の例としては，バグによって関数の再帰呼び出しの深さに歯止めがかからなくなることが挙げられる．データセットに含まれるスタックトレースによると，確かに例外発生時にはスタックに多くのフレームが積まれており，スタックが消費するメモリが大きくなっている．この例外は，スタック領域の境界を超えたアドレスに最初にアクセスしたと思われる MOV，PUSH，CALL などの命令で発生している．この例外によってプログラムが終了するかどうかは，プログラムがこの例外を捕捉して対処するかどうかによる．データセットには，この例外の発生後に長く実行を続ける検体群と，ほどなく実行を終了する検体群の両方が含まれる．

3. 提案システム

IgnorEx の設計と実装について述べる．IgnorEx は Cuckoo を拡張して実装されている．Cuckoo はユーザから解析対象プログラムのファイルを受け取り，それを仮想マシン上のゲスト OS で実行して動作情報をユーザに返す機能を提供する．Cuckoo は解析対象プログラムのプロセスに DLL をインジェクトする．その DLL は API コールの実行や例外の発生を捕捉し，それらの情報をホスト OS で動作する Cuckoo のプロセスに送る．Cuckoo のプロセスはその情報を用いて解析結果を作る．IgnorEx は，この例外の発生を捕捉する部分のコードを拡張して実装されている．その拡張コードはライブラリとして解析対象プログラムのプロセスに動的にリンクされるので，マルウェアプロセスが読み書きできるレジスタとメモリのすべてを読み書きできる．その拡張コードは例外の種類に応じて例外の効果を無効化または改変する処理（以降では**シャドウ例外ハンドラ**と呼ぶ）を実行する．

IgnorEx の現在の実装が捕捉する例外と，それらの例外の捕捉後に実行されるシャドウ例外ハンドラを以下で説明する．これらのどの例外も捕捉後に破棄され，ユーザプログラムには送られない．

ACCESS_VIOLATION 例外: 例外が発生した原因がメモリ読み出し命令の実行であるか，メモリ書き込み命令の実行であるか，命令フェッチであるか，それ以外であるかを調べる．メモリ読み出し命令の実行である場合には，0 や乱数などの事前に決められた値を生成し，それを読み出したデータとみなして命令をエミュレーション実行する．フラグレジスタの値も更新する．メモリ書き込み命令の実行である場合には何も実行しない．すなわち，単にその命令をスキップして次の命令に移る．命令フェッチである場合には現在実行中の関数から即時にリターンさせる．すなわち，スタックの最上フレームをポップし，スタックからリターンアドレスを取り出してそこにジャンプする．関数の返り値として，0 や乱数などの事前に決められた値を生成して格納先レジスタに書き込む．スタックが壊れているなどの理由でリターンが困難な場合には現在実行中の命令列の実行を続ける．すなわち，例外を発生させた命令をスキップして次の命令に移る．なお，解析対象プログラムの関数呼び出し慣例を常に正しくは把握できないため，リターンさせた後のスタックの状態が，本来あるべき状態とは異なるものになることはありうる．

PRIVILEGED_INSTRUCTION 例外: 現在実行中の関数から即時にリターンする．リターンができない場合には，例外を発生させた命令をスキップして次の命令に移る．

ILLEGAL_INSTRUCTION 例外: 現在実行中の関数から即時にリターンする．リターンができない場合には，プログラムカウンタに 1 を足して実行を再開する．

INTEGER_DIVIDE_BY_ZERO 例外: 例外を発生させた除算命令が計算する商と余りとして，0 や乱数などの事前に決められた値を生成し，それらを格納先レ

表 2　実験環境

Host machine	Intel Xeon E5-2620, 128 GB RAM
Host OS	Ubuntu 18.04
Sandbox	Cuckoo Sandbox 2.0.7
Hypervisor	VirtualBox 5.2.18_Ubuntu r123745
Virtual machine	4 cores, 16 GB RAM, 32 GB SATA HDD, host-only network (no Internet access)
Guest OS	Windows 7 Professional SP1 64 bit

ジスタに書き込む．その後，除算命令の次の命令にプログラムカウンタを進めて実行を再開する．

STACK_OVERFLOW 例外： 現在実行している関数から即時にリターンする．

FLOAT_INEXACT_RESULT 例外などの他の例外に対しては，IgnorEx は特別な処理は実行せず，従来より Cuckoo が実行していたログ出力などの処理のみを実行する．その後，例外はユーザプログラムにそのまま配送される．特に，BREAKPOINT 例外と SINGLE_STEP 例外については，マルウェアがデバッガやサンドボックスの検出のためにそれらを意図的に発生させている可能性が十分にあるため，元の挙動を変えないことに大きな意味がある．

現在は，エミュレーション実行は使用頻度が高い命令（MOV，PUSH，POP，TEST，ADD，XOR など）のみに対して実装されている．また，エミュレーション実行においてフラグレジスタを適切に更新する処理も未実装である．エミュレーション実行が未実装である命令で例外が発生したら，その例外の破棄を行い，現在の命令の長さを 2 バイトと仮定して次の命令に移る．

シャドウ例外ハンドラで関数からのリターンを実行する際にスタックが壊れている場合がある．具体的には，ベースポインタやリターンアドレスが不正な値である場合がある．この場合にはリターンをとりやめ，レジスタもメモリも書き換えずに例外の破棄だけを行って実行を再開させる．

現実装では「0 や乱数などの事前に決められた値」としては，rand 関数で生成した 32 ビットの乱数を用いている．

4. 実験

4.1 実験環境

実マルウェアの検体を元の Cuckoo と IgnorEx の上で実行して解析結果を比較した．FFRI Datasets の作成に用いられた検体は非公開であるため，この実験では独自に収集したマルウェアの検体を使用した．それらの検体はすべて Windows 向けの PE 形式の 32 bit バイナリプログラムであり，Cuckoo 上での実行で最低 1 回は例外を発生させる．検体数は 861 である．実験環境を**表 2** に示す．ゲスト OS の起動モードは headless に設定した．解析のタイムアウト時間は 60 秒に設定した．

4.2 挙動の変化の度合いと検体数の関係

まず，Cuckoo と IgnorEx の上でのマルウェアの挙動を比較することにより，IgnorEx がどの程度の割合のマルウェアの実行をどの程度変化させるかを調べた．比較結果を**表 3** に示す．表中の「実行された API コール数」としては，各マルウェア検体の全プロセスを通じた API コール数の和を使っている．また，「アクセスされた資源の数」としては，Cuckoo による解析結果の behavior の部分に含まれる項目（オープンや読み出しなどの操作ごとの，アクセスされたファイルパスやレジストリキーなど）の数を使っている．それらの数はどの程度多くの挙動を引き出せたかを測る目安になると考えている．どちらかの数が元々 0 である検体は実験対象から外している．Cuckoo による解析結果では例外は API コールと共通の形式で記述されているが，ここでは API コールの数には例外は含めていない．これらの数は検体によって値が大きく異なり，また，両システム上での解析によるそれらの数の比も，1:0 から 1:10000 以上までと極めて多様である．よって，それらの数や比の平均値にはほとんど意味がない．よってここでは，Cuckoo を IgnorEx に置き換えることにより，それらの数が大きく変化した検体の割合と，ほとんど変化しなかった検体の割合を示すことにしている．

表 3　提案手法の導入による変化の大きさの観点で検体を分類したときの，各分類に含まれる検体の数と割合

	実行された API コール数		アクセスされた資源の数	
10 倍以上に増加	9	(1.0%)	106	(12.3%)
2 倍以上 10 倍未満に増加	12	(1.4%)	178	(8.4%)
1.05 倍以上 2 倍未満に増加	22	(2.6%)	42	(4.9%)
1 倍以上 1.05 倍未満に増加	215	(25.0%)	309	(35.9%)
0.95 倍以上 1 倍未満に減少	192	(22.3%)	4	(0.5%)
0.5 倍以上 0.95 倍未満に減少	123	(14.3%)	65	(7.5%)
0.1 倍以上 0.5 倍未満に減少	107	(12.4%)	75	(8.7%)
0.1 倍未満に減少	181	(21.0%)	188	(21.8%)

IgnorEx の導入により，30.0%の検体で API コール数が増加し，61.4%の検体でアクセスされる資源の数が増加した．また，12.3%の検体で，アクセスされる資源の数が 10 倍以上に増加した．これらの結果は，マルウェアに多くの API コールの実行や多くの資源へのアクセスをさせる点で IgnorEx は有用であることを示していると考える．とはいえ，API コール数については，増える検体よりも減る検体のほうが多い．また，これらの数の変化が小さい範囲（0.95 倍から 1.05 倍）にとどまる検体の割合も，それぞれ 47.3%と 36.4%もあり，少数派ではない．さらに，数が 0.1 倍未満と大きく減る検体も共に 21%以上あった．現状では，IgnorEx を導入してもマルウェアの挙動がほとんど変化しない可能性や，引き出せる挙動が減る可能性もあることは認識する必要がある．現状では，IgnorEx と元の Cuckoo を相互補完的に併用することが，多くの挙動情報を得る上で効果的である．

なお，IgnorEx 上では一部の検体の実行で例外の発生回数が極めて多くなる現象が観測されている．これらの検体

の多くの実行においては，全く同じか非常に似た例外が繰り返し発生する．この現象が起こる理由の 1 つは，IgnorEx のシャドウ例外ハンドラにより，解析対象プログラムが例外を発生させる無限ループを実行するようになることである．IgnorEx 上での実行では，元の Cuckoo 上で実行終了をもたらした例外が発生しても，それらは多くの場合に実行終了をもたらさない．また，例外を発生させた命令がアクセスするメモリ領域がカウンタなどのループ終了条件に関わる値を保持している場合には，その命令の挙動が改変されることにより永久に終了条件が満たされなくなりうる．このような場合には，解析のタイムアウト時間まで，例外を発生させる無限ループを実行することになる．

4.3 ケーススタディ： 個々の検体における挙動の変化

いくつかの検体を取り上げて挙動を説明し，IgnorEx によって従来観測できなかった挙動が観測できるようになることと，それができない検体もあることを示す．

4.3.1 検体 A

この検体は Cuckoo 上では API コールとして GetSystemTimeAsFileTime のみを実行した．その後に不正なメモリアドレスからの読み出しを試みて ACCESS_VIOLATION 例外を発生させ，直後に実行を終了した．よって，ファイルもレジストリも全くアクセスしていない．Cuckoo 上での動的解析だけからでは，この検体の挙動をほとんど全く知ることができなかった．

IgnorEx 上での実行では，上記の例外はシャドウ例外ハンドラによって破棄され，例外を発生させた命令が値をセットすべきレジスタには乱数がセットされた．実行再開後にも同じコードアドレスで同じアドレスに読み出しを試みて，繰り返し同じ例外が発生した．例外が 479 回発生した後に繰り返しは終了し，その後はファイルやレジストリなどの資源に対して多くの操作を行った．最終的には 9 種類のファイルのオープンを試み，そのうち 4 種類のファイルのオープンに成功した．オープンに成功したファイルと失敗したファイルにはそれぞれ例えば `win.ini` と `PINBALL.DAT` がある．さらに，54 種類のレジストリキーの読み出しと 5 種類のレジストリキーへの書き込みを実行した．アクセスされたレジストリキーには例えば `SpaceCadet` や `wdmaud.drv` があった．IgnorEx の導入により，新たに多くの挙動を引き出すことができた．

4.3.2 検体 B

この検体は Cuckoo 上では 177 回の API コールの後，不正なメモリアドレスからの読み出しを試みて ACCESS_VIOLATION 例外を発生させた．その後，FLOAT_INEXACT_RESULT 例外を 6 回発生させた．どの例外の後にも実行は終了せずに続いた．その後，OUT 命令の実行で PRIVILEGED_INSTRUCTION 例外を発生させた．この例外発生後，資源を解放してプロセスの実行を終了させる API コール列を実行した．結局，API コールは 249 回実行され，API コールの引数に与えられたファイルパスは 5 種類，レジストリキーは 6 種類だった．

IgnorEx 上では，上記の ACCESS_VIOLATION 例外に対してシャドウ例外ハンドラが実行された後，API コールや発生する例外が Cuckoo 上でのそれとは異なるものに変化した．その後，OUT 命令ではなく CLI 命令で PRIVILEGED_INSTRUCTION 例外が発生したが，シャドウ例外ハンドラが例外の破棄と関数からのリターンを行い，実行は継続した．結局，API コールを 1654 回実行し，様々な資源を操作した．API コールの引数に与えられたファイルパスは 41 種類，レジストリキーは 118 種類だった．Cuckoo 上での実行では現れなかった多くの操作が行われた．それらの操作には `C:\Windows` への `xk.exe` というファイルの作成，自身のファイルの `C:\Windows\System32\IExplorer.exe` へのコピー，`ComputerName` というレジストリキーの読み出しなどが含まれる．

4.3.3 検体 C

この検体は，IgnorEx を導入しても挙動に関する追加情報をほとんど得られなかった検体の例である．この検体は Cuckoo 上では API コールを 1 回も実行しないまま，ACCESS_VIOLATION 例外を発生させて実行を終了した．例外発生時にプログラムカウンタが指すメモリ領域には非常に多くの 0 が続いていた．x86 で 0x00 0x00 という機械語コードに対応するのは add byte [eax], al という命令である．この命令の実行時に EAX レジスタに入っていた値はアクセス可能なメモリ領域のアドレスではなかったため，例外が発生したと考えられる．多くの 0 が続く機械語コードは明らかに正しい命令列ではないので，この検体は何らかの理由で実行に異常をきたし，命令列ではないメモリ領域にジャンプした可能性が高い．

IgnorEx 上でのこの命令の実行では，EAX レジスタが保持するアドレスに対しては読み出しも書き込みも行わずに次の命令に移る．メモリ領域には 0 が続くため，上記の ADD 命令を実行して例外を発生させては次の命令に移ることを 1900 回以上繰り返した．その後，0 の続く領域が終わり，0xff 0xff を読み出して ILLEGAL_INSTRUCTION 例外を発生させた．IgnorEx は実行中の関数から即時にリターンして実行を再開させたが，直後にこの検体は NtTerminateProcess を実行して終了した．

5. 議論

無限ループ： 例外の効果を無効化，改変し続けると，プログラムは終了のための API 関数を通じて自ら実行を終了するか，無限ループに陥るかのどちらかになる．実際，IgnorEx 上での実行では，例外で終了しなくなったことにより無限ループに陥ったと予想される検体が多数存在した．現実装ではタイムアウトによる強制終了を利用してそ

のような検体を終了させているが，タイムアウト時間を長くするなどの場合には，無限ループの実行の検出が必要になる可能性がある．そのための拡張としては，例えば，同じ場所で同じ種類の例外が所定の回数発生したら解析を打ち切るというものがある．

セマンティクスの改変： シャドウ例外ハンドラはプログラムのセマンティクスを変えうる．本研究では，そのような処理を通じて得られた動的解析結果をどの程度信頼するかについてはユーザに委ねている．直感的には，アクセスされることがIgnorExによって新たにわかるファイルなどの資源は，元々検体がアクセスを意図していたと推測するが，確実な裏付けがあるわけではない．理想的には，解析結果の各部分の信頼度についての情報を解析システムが提供できれば望ましいと考える．

6. 関連研究

IgnorExがACCESS_VIOLATION例外に対して行う処理はFailure-Oblivious Computing (FOC) [6]の研究で既に提案されている．FOCは改造Cコンパイラによるコード挿入によって当該処理を実現するが，IgnorExではサンドボックスがプロセスの状態を書き換えて実現する．マルウェアのソースコードは通常は入手できないため，IgnorExのようにマルウェアを対象とするシステムは，ソースコードを必要としないことが望ましい．また，FOCでは不正なメモリアクセスのみに対処するが，IgnorExでは特権命令の実行などの他の例外的な処理にも対処する．

Fix2Fit [3]は，クラッシュを発生させるソフトウェアの誤りを修正するために複数のパッチ候補を生成し，クラッシュを発生させないパッチをテストによって選ぶシステムである．本研究はバイナリプログラムが対象であり，彼らの研究とは前提も手法も異なるが，クラッシュしない実行を自動的に得ることを狙う点では共通している．複数の候補から良いものを選ぶ彼らの手法は，本研究の手法でも有効である可能性が高い．

Rx [5]は，障害を検知したら，ソフトウェアの状態をロールバックしてメモリ管理や非同期的実行に関する動作を変えて再実行することにより，その障害が発生しない実行を提供するシステムである．彼らはマルウェアを用いた実験は行っていないが，マルウェアから多様な挙動を引き出すためにRxの手法は有効である可能性がある．

ソフトウェアのクラッシュ時のログやダンプから，クラッシュの原因や原因となったプログラムの場所を特定するシステムの研究が存在する．例えばRETracer [1]やPOMP [7]があるが，これらはいずれもマルウェアのクラッシュを対象とした研究ではなく，ソフトウェアの実行終了を防ぐ手法を扱った研究でもない．

ReBucket [2]は，クラッシュレポートを原因特定などのためにクラッシュ時のスタックの類似性に基づいて分類するシステムである．提案手法では例外コードのみに基づいて例外を分類して実行すべき処理を決定したが，この研究のように別の情報も用いて例外発生原因をより詳細に特定すれば，よりきめ細かい処理を実行できる可能性がある．

7. まとめと今後の課題

本研究では，マルウェアが例外を発生させたときにその状態を改変して実行を継続させる手法を提案した．また，それに基づくシステムを実装してマルウェアを解析し，既存のシステムでは観測できなかった多くの挙動をそのシステムによって観測できるようになったことを確認した．

今後の課題としては，まず，解析対象プログラムが無限ループに陥って新たな挙動を示さなくなる問題を解決する必要がある．また，提案手法は解析対象プログラムのセマンティクスを変えうるため，そうして得られた解析結果の信頼性や有用性については今後さらに研究が必要である．

謝辞 FFRI Datasetsを提供していただいた（株）FFRIおよびMWS組織委員会に感謝する．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] Cui, W., Peinado, M., Cha, S. K., Fratantonio, Y. and Kemerlis, V. P.: RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps, *Proceedings of the 38th International Conference on Software Engineering*, pp. 820–831 (2016).

[2] Dang, Y., Wu, R., Zhang, H., Zhang, D. and Nobel, P.: ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity, *Proceedings of the 34th International Conference on Software Engineering*, pp. 1084–1093 (2012).

[3] Gao, X., Mechtaev, S. and Roychoudhury, A.: Crash-avoiding Program Repair, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 8–18 (2019).

[4] Oyama, Y.: How Does Malware Use RDTSC? A Study on Operations Executed by Malware with CPU Cycle Measurement, *Proceedings of the 16th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, LNCS, Vol. 11543, pp. 197–218 (2019).

[5] Qin, F., Tucek, J., Zhou, Y. and Sundaresan, J.: Rx: Treating Bugs As Allergies—A Safe Method to Survive Software Failures, *ACM Transactions on Computer Systems*, Vol. 25, No. 3 (2007).

[6] Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T. and William S. Beebee, J.: Enhancing Server Availability and Security Through Failure-Oblivious Computing, *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pp. 303–316 (2004).

[7] Xu, J., Mu, D., Xing, X., Liu, P., Chen, P. and Mao, B.: POMP: Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts, *Proceedings of the 26th USENIX Security Symposium*, pp. 17–32 (2017).

[8] 荒木粧子，笠間貴弘，押場博光，千葉大紀，畑田充弘，寺田真敏：マルウェア対策のための研究用データセット～MWS Datasets 2019～，情報処理学会研究報告，Vol. 2018-CSEC-86 (2019).