

2F1-3

機械語命令フックによる 耐解析機能の特性を利用したマルウェア対策

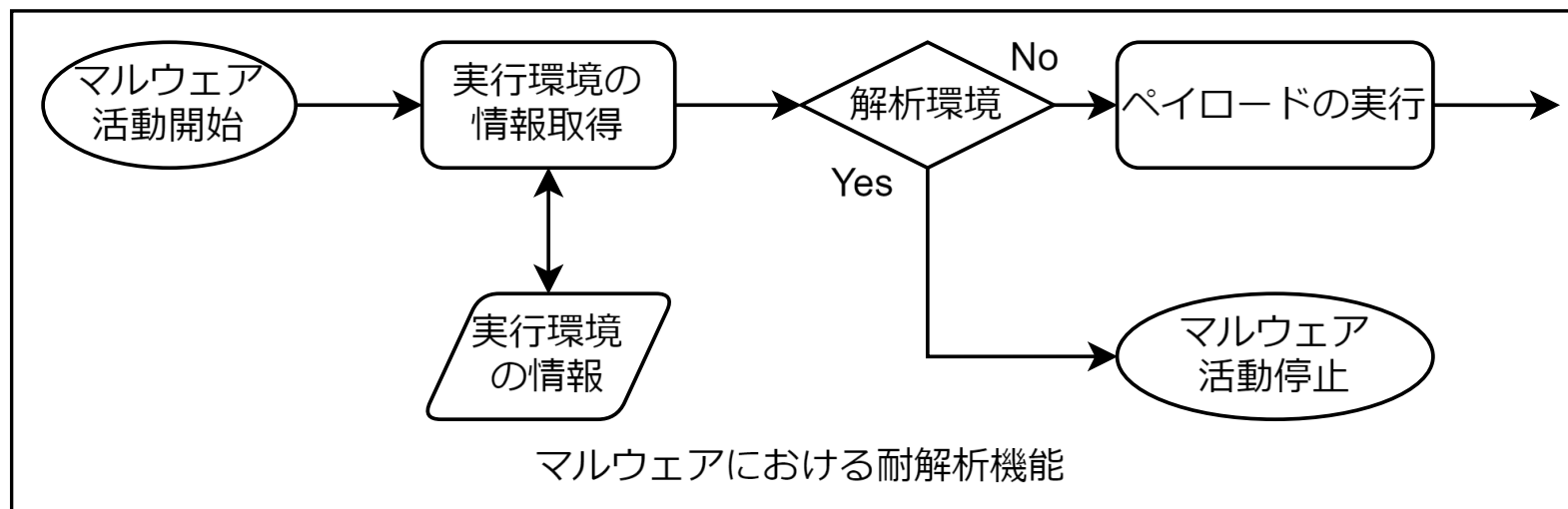
2025/01/29

©小松 蒼樹*, 毛利 公一*, 瀧本 栄二*†

*立命館大学, †奈良女子大学

はじめに(1/2)

- 企業などによるテレワークの導入&マルウェアによる侵入経路の多様化によりエンドポイントセキュリティが必要不可欠である
- 多くのマルウェアには耐解析機能が実装されている



- 耐解析機能の性質を利用した端末保護手法が提案されている[1]
 - ✓ 解析環境を示す偽の情報を与えることで、マルウェアの活動停止に導く

[1] 松木隆宏,新井悠,寺田真敏,土居範久,マルウェアの耐解析機能を逆用した活動抑止手法の提案,情報処理学会論文誌,Vol.50,No.9, 2118-2126,(2009).

■耐解析機能にはさまざまな手法が存在

✓ 検知対象別

- デバッガ(Anti-Debug)
- 仮想環境(Anti-VM)

✓ 検知手法別

- Windows APIを利用
- 特定の機械語命令を利用
- 直接メモリを参照

■上記の中で、特定の機械語命令を利用する手法に着目

- ✓ 関連研究で扱われていない
- ✓ 実際のマルウェアで一定数観測[2]



**Windows上のターゲットプロセスをデバッガの制御下にする事で、
機械語命令を用いる耐解析機能に偽情報を与え、マルウェアの活動抑止**

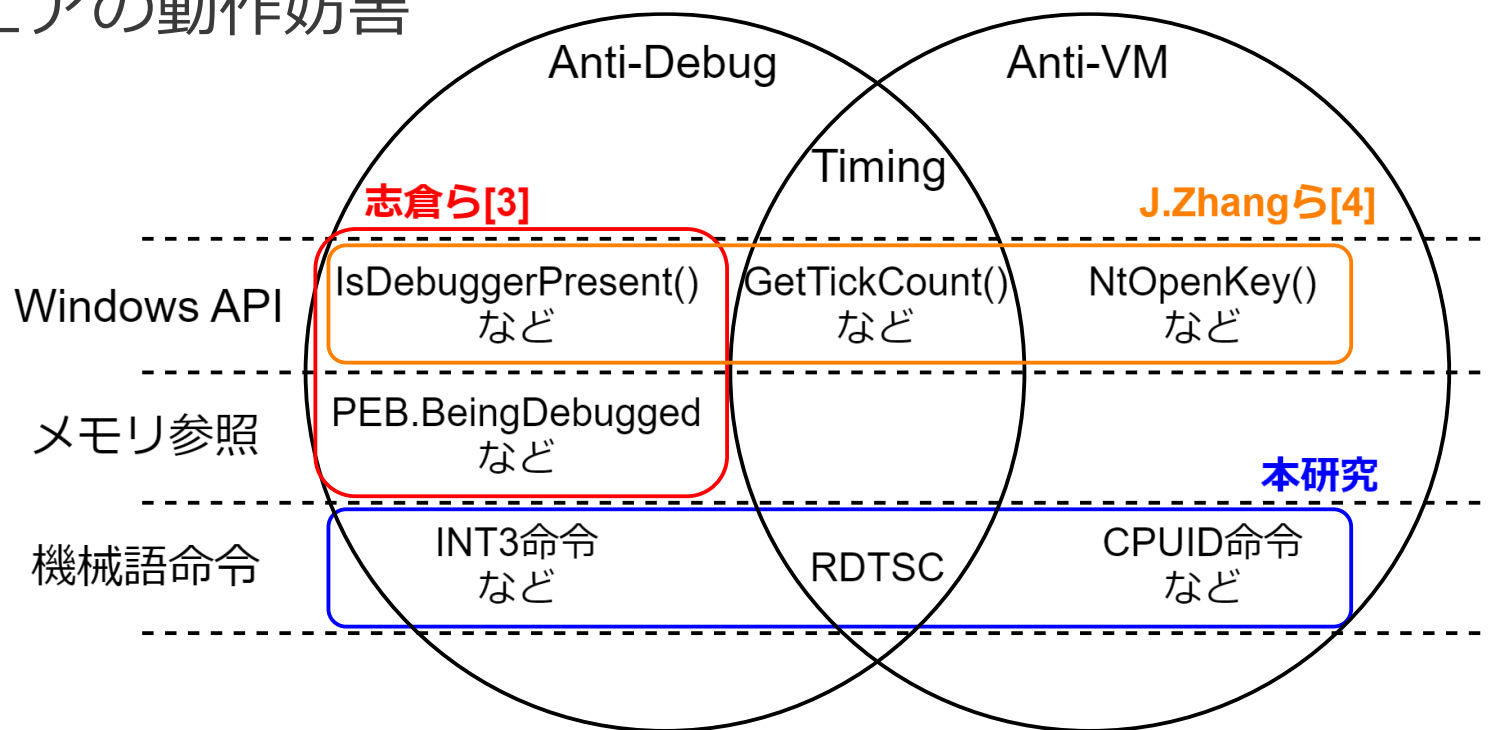
[2] Nicola Galloro, Mario Polino, Michele Carminati, Andrea Continella, Stefano Zanero, A Systematical and longitudinal study of evasive behaviors in windows malware, Computers & Security, Volume 113, 2022, 102550,

■志倉ら[3]

- ✓実際にデバッガとして認識される
プロセスを生成し, マルウェアの動作妨害

■J.Zhangら[4]

- ✓Windows APIをフックする
ことで欺瞞環境を構築



[3] 志倉大貴,西村俊和,瀧本栄二,軽量デバッガを用いたマルウェア動作妨害機構の実装と評価,コンピュータセキュリティシンポジウム 2022,24-27,(2022).

[4] J.Zhang et al., "Scarecrow: Deactivating Evasive Malware via Its Own Evasive Logic", 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN),Valencia,Spain,2020,76-8

機械語命令を利用した耐解析機能と今回の対象手法

■機械語命令を利用した耐解析機能は複数手法存在

- ✓今回は、CPUID命令(EAX=0x1)の手法に対して機械語命令フックを用いて偽情報を与えるプロトタイプを実装した

■デバッガを検知(Anti-Debug)

機械語命令	実行直前のレジスタ値	仮想環境の検知方法
CPUID命令	EAX = 0x1	ECXレジスタ31bit目(hypervisor bit)
	EAX = 0x40000000	ECX, EDXの仮想化ベンダ文字列
IN命令	EAX = 'VMXh' / EDX = 'VX'	EBX = 'VMXh'になる (VMWareの検出)
SIDT,SGDT,SLDT命令		実環境とVMの記述子テーブルのアドレスの違い

■仮想環境を検知(Anti-VM)

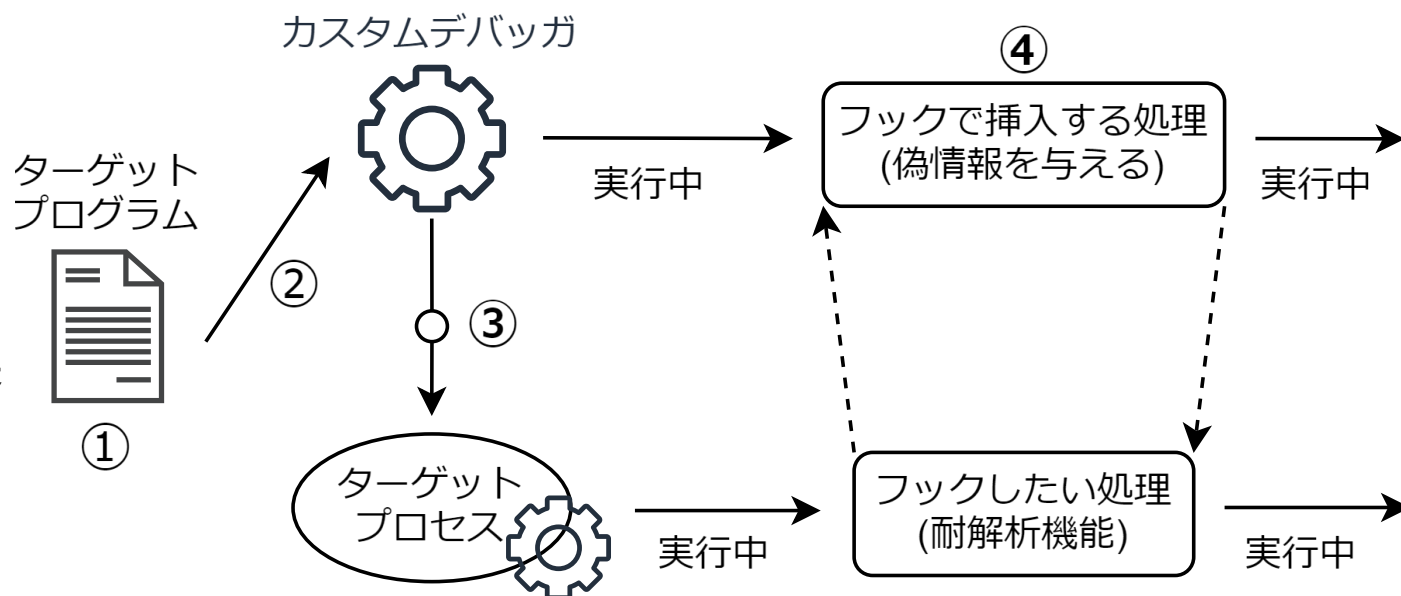
機械語命令	デバッガの検知方法
INT3	例外の通知有無

■ターゲットとなるプロセスに対してデバッガを実際にアタッチ

- ✓デバッガを用いる事で機械語命令をフックし，耐解析機能に解析環境を示す偽の情報を与える

■提案手法の流れ

1. ターゲットの起動指示
2. デバッガを代わりに起動
3. ターゲット生成後，実行前に機械語命令を探索しBPで置換
4. BP発生時に偽情報を与える



▼

フックで与えた偽情報により，耐解析機能に実行環境を誤認させることが可能

■機械語命令フックにより，耐解析機能に偽情報を与えられる

- ✓ 解析環境と誤認させることでマルウェアのペイロードを実行させない

■実際にターゲットとなるプロセスにデバッガをアタッチする

- ✓ 多くのAnti-Debugに対して有効である

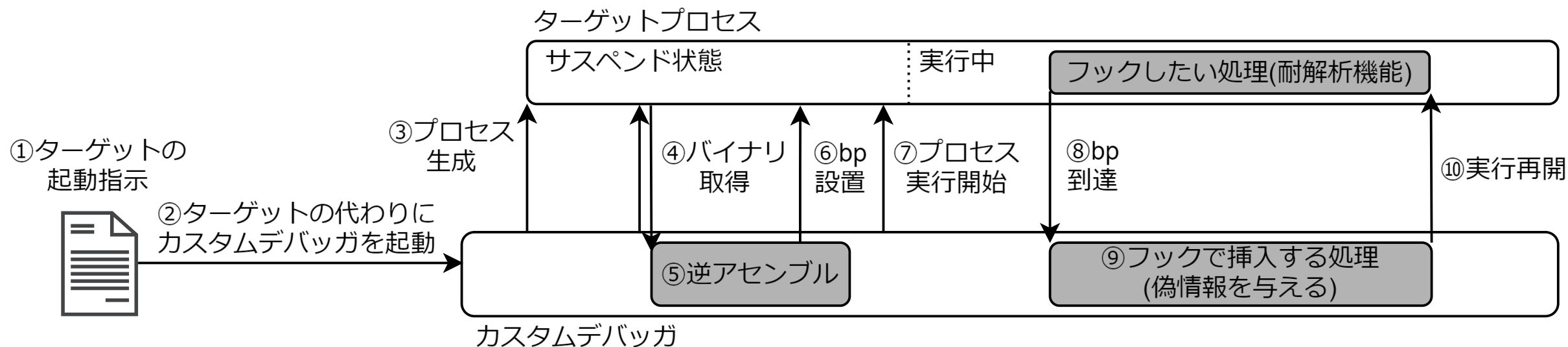
■命令列を実行中に監視する事による，処理時間の増加を防げる

- ✓ 動的計装を用いたフックなどでは，実行命令の全てにおいて監視される
- ✓ 実行命令数が増加するとオーバーヘッドも比例して増加する



- ✓ 本手法では実行前にフックポイントを指定する為，実行中におけるフック箇所以外のオーバーヘッドを削減可能

■実装の全体図



■今回のプロトタイプ実装では、ターゲット上のCPUID命令を対象

- ✓ カスタムデバッガは、CPUID命令上にブレークポイントを設置
- ✓ ブレークポイント到達時のEAXレジスタ値が0x1の場合のみ、偽情報を与える
 - CPUID命令を用いたAnti-VMは、EAX=0x1の状態で実行される特徴から



①～③ターゲットプロセスの生成

■ターゲットプロセス生成までにデバッガを起動する

- ✓ 本手法では、ターゲットの起動指示後に代わりにカスタムデバッガを生成し、カスタムデバッガからターゲットプロセスを生成する
- ✓ Image File Execution Options(IFEО)を用いてこの仕組みを実現

■Image File Execution Options

- ✓ プロセス生成処理中に参照されるレジストリであり、設定した実行可能ファイルのプロセスが生成される際に、代わりにデバッガを起動させることが可能
 - 例) HKLM¥SOFTWARE¥Microsoft¥Windows NT¥CurrentVersion¥Image File Execution Options¥Notepad.exe

名前	種類	データ
 (既定)	REG_SZ	(値の設定なし)
 debugger	REG_SZ	C:¥Users¥skomatsu¥Desktop¥ProtectionDebugger.exe

■起動したデバッガ処理内でターゲットプロセスを生成

[5] Image File Execution Options, <https://learn.microsoft.com/ja-jp/previous-versions/windows/desktop/xperf/image-file-execution-options>, (参照:2025/1/22)

④～⑥CPUID命令上にブレークポイント設置

■CPUID命令上にブレークポイントを設置する

- ✓ ソフトウェアブレークポイント(INT3命令)をCPUID先頭アドレスに書き込む

0x00: mov rax,0x1

0x07: cpuid **INT3**

0x09: test rcx 0x80000000

■CPUID命令の先頭アドレスを取得する必要がある

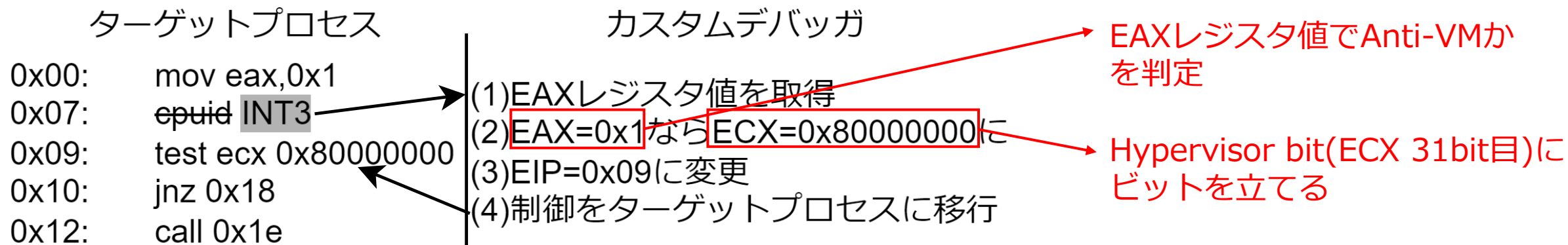
- ✓ バイナリを取得後, 逆アセンブルすることでアドレスを取得する
 - バイナリ
 - main関数アドレスから50000バイトを取得する
 - PEファイルフォーマット.AddressOfEntryPoint + PEB.ImageBaseでアドレスを取得
 - 逆アセンブル
 - 逆アセンブラとしてudis86[6]を利用する
 - 軽量でかつC言語コードに直接組み込み可能な逆アセンブラライブラリ

[6] vmt/udis86: Disassembler Library for x86 and x86-64, <https://github.com/vmt/udis86>

⑧⑨ブレークポイント到達後に挿入する処理

■bpに到達するとターゲットプロセスの制御がデバッガに移行する

- ✓ EAX=0x1時のCPUID命令を用いたAnti-VMに偽情報を与える

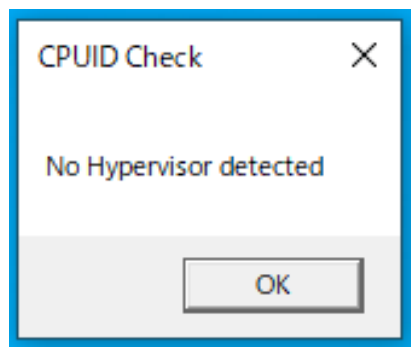


本手法適用時におけるCPUID命令の実行結果

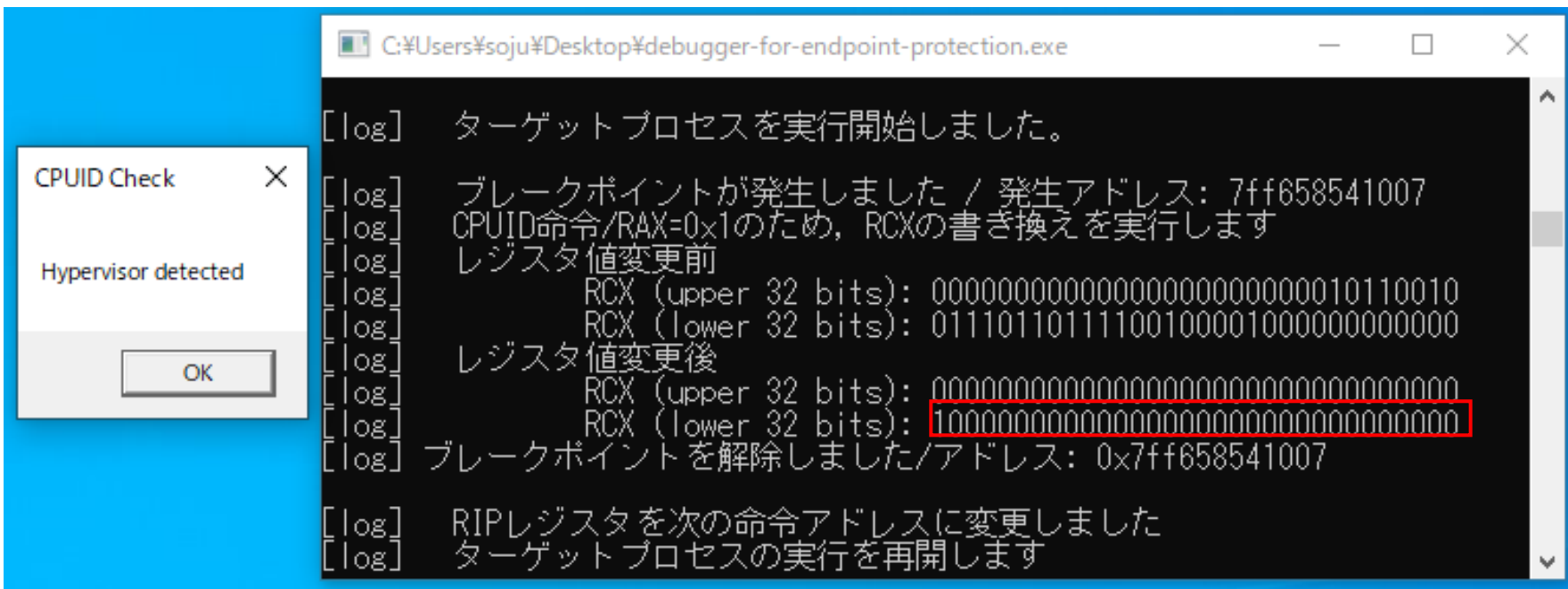
■CPUID命令(Anti-VM)を実装したプログラムを作成

- ✓ CPUID命令を用いたAnti-VMを実行し，判定結果をMessageBoxA関数で表示

■プログラム実行結果



通常起動時のAnti-VM結果



本手法適用時のAnti-VM結果とデバッガの出力結果

■オーバヘッド計測の目的

- ✓ ターゲットプロセスの処理内容に関わらず発生するオーバヘッドを確認
 - ターゲットプロセスを生成するまでの時間
 - CPUID上にブレークポイントを設置するのに必要な時間
- ✓ デバッガに処理が移行する際のターゲットプロセスの処理停止時間の確認
 - デバッグイベント発生時
 - CPUID上に設置したブレークポイント発生時

■オーバヘッド計測に用いた環境

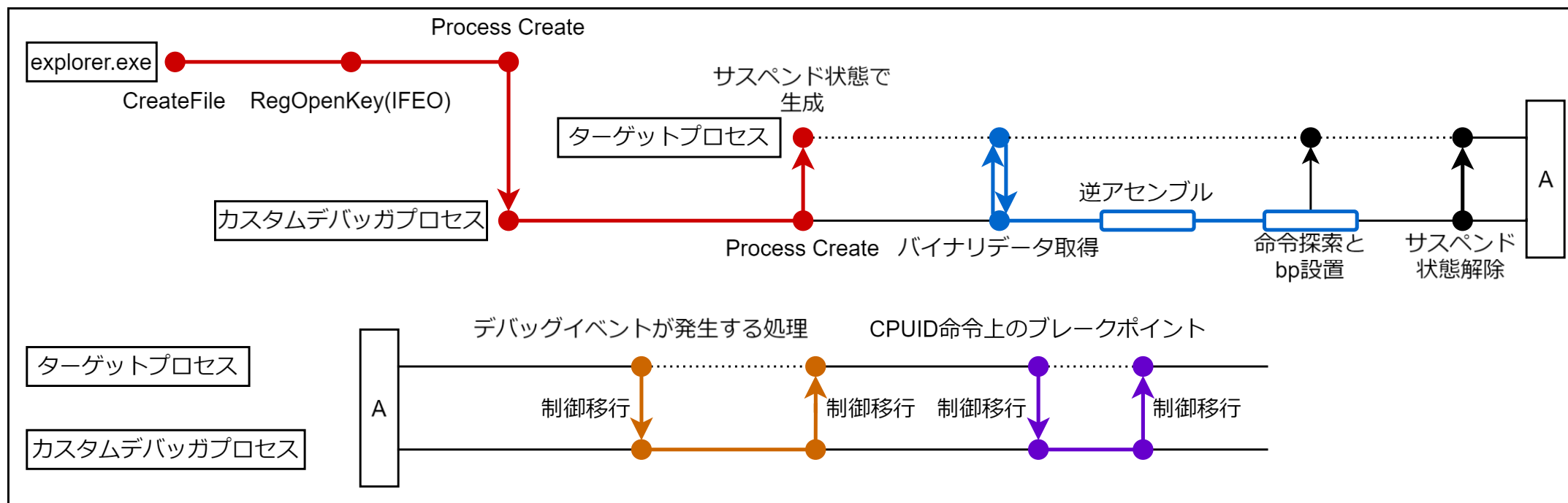
- ✓ Process Monitorとターゲットプロセス,
デバッガを起動した状態で計測

要素	計測環境
CPU	Intel Core i7-12700K 3.61GHz
メモリ	32.0GB
OS	Windows 10 Education 22H2

[8] プロセスモニター – Sysinternals,
<https://learn.microsoft.com/ja-jp/sysinternals/downloads/procmon>, (参照:2025/1/22)

■以下の各区間においてそれぞれ10000回計測し，平均値を算出

- ✓ プロセス起動指示からターゲットプロセス生成まで(1)
- ✓ バイナリ取得～bp設置にかかる区間(2)
- ✓ デバッグイベント発生処理の実行時間とターゲットに制御が戻るまで(3)
- ✓ CPUID上に設置したbpで処理移行した後に制御がターゲットに戻るまで(4)



■バイナリ取得～bp設置にかかる区間(2)

- ✓ 50,000バイトのバイナリを取得し, 逆アセンブル後にbpを設置している
- ✓ 計測プログラムの50,000バイト中に3つのCPUIDが存在
 - 2つのCPUID命令はコンパイラによって設置, 1つはプログラム上で記述
 - 3つ全てのCPUID命令にbpを設置

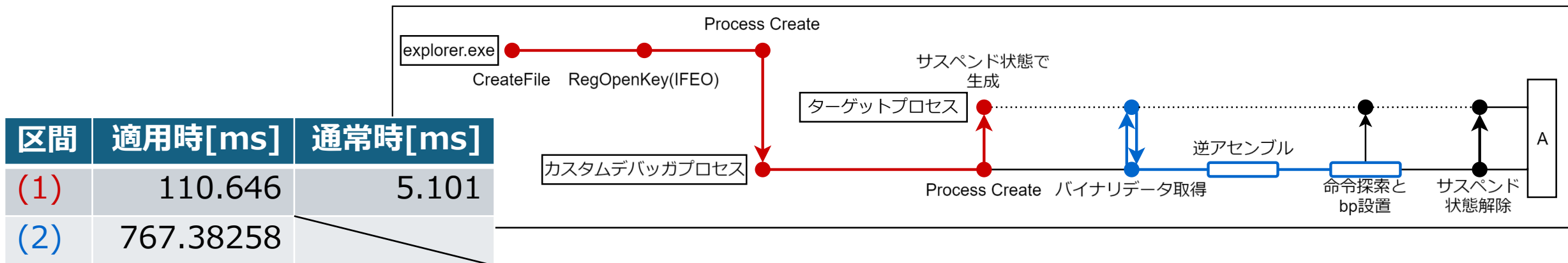
■各デバッグイベント発生後にターゲットに制御が戻るまでの区間(3)

- ✓ 以下のデバッグイベントは未計測
 - CREATE_PROCESS, EXIT_PROCESS: 複数プロセス間で正確な計測が困難なため
 - RIP_EVENT: 発生する確率が低いため

サスペンド解除前の区間におけるオーバヘッド計測結果

毛利研究室

- Process Monitor[8]にて，ログに記録されたTimeの差分にて算出(1)
- デバッガ処理内にて，C++標準ライブラリchronoを用いて計測(2)

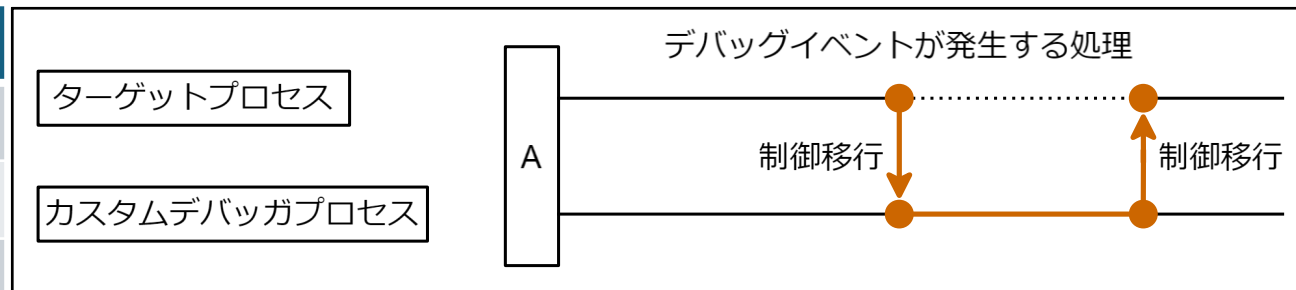


- 区間(1):IFE0を用いる事で約21.7倍の時間を要する事を確認
- 区間(2):約767msが通常時と比べて増加し，逆アセンブルするバイナリサイズに比例するため，サイズを最適に抑える工夫が必要

デバグイベント発生時のオーバーヘッド計測結果

■デバグ対象プロセスにてC++標準ライブラリchronoを用いて計測

デバグイベント	通常時[μs]	適用時[μs]
Load_DLL	4.3053	4.3083
UnLoad_DLL	0.1201	0.1193
Create_Thread	19.7072	25.0617
Exit_Thread	5.6845	10.1869
Output_Debug_String	20.7371	75.5958
Exception(Breakpoint)		980.8297



```
auto startLoad = std::chrono::high_resolution_clock::now();  
HMODULE hModule = LoadLibraryW(dllPath);  
auto endLoad = std::chrono::high_resolution_clock::now();
```

ターゲットプロセス上の計測コード

■各デバグイベントがターゲットプロセスで発生すると、上記のオーバーヘッドが発生する

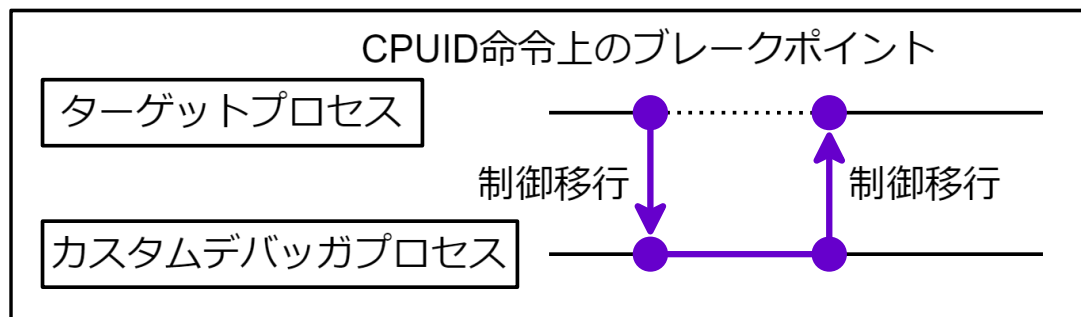
- ✓ デバグイベントの発生回数などはプロセスによって異なる為、デバグイベントによって発生するオーバーヘッド合計を予測することは難しい

CPUID命令上のbp発生時のオーバヘッド計測結果

■デバッグ対象プロセスにてC++標準ライブラリchronoを用いて計測

✓CPUID命令を生成する__cpuid()の処理時間を計測

区間	適用時[μs]
(4)	517.4332



```
INT CPUInfo[4] = { -1 };  
auto startLoad = std::chrono::high_resolution_clock::now();  
__cpuid(CPUInfo, 1);  
auto endLoad = std::chrono::high_resolution_clock::now();
```

ターゲットプロセス上の計測コード

- 約517μsの間ターゲットプロセスの処理が停止するため、CPUID命令の数が増加すると停止回数が増加し、オーバヘッドが増加

■ターゲットプロセスの処理内容に関わらず発生するオーバヘッド

✓プロセス生成まで(1)

- IFEOを用いてデバッガからターゲットプロセスを起動すると、通常起動時の約21.7倍のオーバヘッドがかかる

✓バイナリ取得～bp設置まで(2)

- 約767msが通常時と比べて増加
- 逆アセンブルするバイナリサイズに比例するため、サイズを最適に抑える工夫が必要

■ターゲットプロセスによって発生回数が増加するオーバヘッド

✓デバッグイベント発生時(3)

- デバッグイベントが発生すると、未適用時と比べ、オーバヘッドが増加する
- デバッグイベントによって発生するオーバヘッド合計を予測することは難しい

✓CPUID上に設置したbp発生時(4)

- 約517 μ sの間、デバッグ対象プロセスの処理が停止する

■バイナリ取得～bp設置にかかる区間(2)のオーバヘッドの削減方法

✓バイナリ数を削減

- 耐解析機能は、解析を防止する目的で実装されるためマルウェア処理の前半に実装されていると想定できる
- 対象にするバイナリを、マルウェア処理の前半に絞ることでバイナリ数を削減可能になりオーバヘッド削減が見込める

✓区間(2)の実行タイミングを変更

- 区間(2)の処理を実行可能ファイルがメモリ上にロードされた後に実行している
- ロード前に予め該当処理を実行することでサスペンド状態解除までの時間を削減

- **カスタムデバッガを用いて機械語命令を用いる耐解析機能をフックし、解析環境を示す偽の情報を与える**
 - ✓ 偽の情報を与える事で解析環境であると誤認させ、ターゲットプロセスの活動を抑止する
- **CPUID命令のAnti-VMに対し本手法を適用したプロトタイプを実装**
 - ✓ CPUID命令のAnti-VMを実装したテストプログラムを用いることで、本手法の有効性を確認
 - ✓ 各オーバーヘッド計測し、削減方法について考察
- **今後は、CPUID命令以外の耐解析機能についても実装を進めていく**