Computer Security Symposium 2018
22 - 25 October 2018

マルウェアによるRDTSC命令の利用方法についての分析

大山 恵弘[1]

概要：多くのマルウェアが解析を回避するための処理を実行する．それらの中には，実行時間や実行 CPU サイクル数の計測によるサンドボックス検出があり，特に，RDTSC 命令を用いる手法が広く知られている．しかし，実際のマルウェアが RDTSC 命令をどう利用しているかの実態は十分に明らかにされてこなかった．本稿では，マルウェアによる RDTSC 命令の利用方法を分析した結果を示す．この分析では，マルウェアの命令列から RDTSC 命令の周辺のコード断片を抽出し，それらを特徴にしたがって分類した．その結果，マルウェアが RDTSC 命令により多様な処理の CPU サイクル数を計測していることや，処理に要する CPU サイクル数の計測以外の目的で RDTSC 命令を実行している可能性があることがわかった．

キーワード：マルウェア，RDTSC 命令，解析回避，耐解析，サンドボックス，仮想化

Analysis on the Usage of the RDTSC Instruction by Malware

Yoshihiro Oyama[1]

Abstract: Many malware programs execute operations for evading analysis. They include sandbox detection by measuring execution time or executed CPU cycles, and in particular, a method of using the RDTSC instruction is widely known. However, the actual usage of RDTSC by real malware has not yet been sufficiently clarified. In this paper, we show the analysis result of the usage of RDTSC by malware. In this analysis, code fragments around RDTSC are extracted from the instructions of malware programs, and classified based on their characteristics. The result shows that malware programs measure the number of CPU cycles of diverse operations and may execute RDTSC for purposes other than the measurement of the number of CPU cycles required for operations.

Keywords: Malware, RDTSC instructions, analysis evasion, anti-analysis, sandbox, virtualization

1. はじめに

最近のマルウェアの多くは自身の解析を回避するための処理（解析回避処理）を実行する．解析回避処理には様々な種類があるが，よく知られた有効な処理の 1 つは，性能情報を利用するものである．マルウェアは特定の処理の実行にかかる時間やサイクル数を計測し，それが異常に長い場合や多い場合には，自身がサンドボックスや仮想マシンの上で動いている可能性があると判断し，実行を終了するなどの対策をとる．性能情報としては，RDTSC 命令により取得される time stamp counter (TSC) が利用可能である．TSC は CPU の中にあるカウンタであり，1CPU サイクルごとに 1 増加する．特定の処理の実行の前後で TSC を取得して差を計算することにより，その実行に要したサイクル数を高い精度で知ることができる．

RDTSC 命令を用いて解析を回避する方法やその効果に関しては多くの研究 [3, 6, 11–14, 16–18] が行われ，その方法を用いたソフトウェア [1, 10] も開発されてきた．例えば，CPUID 命令の実行に要するサイクル数を調べる方法が，仮想マシンの有無を推定する上で有効であることが明らかにされてきた [3]．実際のマルウェアでもサイクル数の計測による解析回避は用いられており，例えば，`push eax` 命令の実行に要するサイクル数を測って解析を回避するマルウェアが存在する [14]．また，2 種類の Windows API

[1] 筑波大学
University of Tsukuba

の実行の前後で RDTSC 命令を実行して解析を回避しようとするマルウェアも知られている [5]．マルウェアを解析，検知する著名なツールである YARA にも，RDTSC 命令と CPUID 命令の実行を検出するためのルールが存在する．

しかし，実際のマルウェアが RDTSC 命令をどう利用しているか，RDTSC 命令を通じて何を実現しようとしているかについては，十分に明らかにされているとは言い難い．例えば，CPUID 命令の利用が仮想マシン検出に有効であることは指摘されているものの，実際のマルウェアの多くがその方法を用いているかどうかは不明確である．マルウェアによる解析回避を防ぐためには，マルウェアが実行しようとしている解析回避処理を正しく把握し，それを無効化するための適切な対策を実行する必要がある．

そこで，本研究では，実際のマルウェアが RDTSC 命令を利用して何をしているかの実態を明らかにする．この知見により，マルウェアによる RDTSC 命令を用いた解析回避の試みについての最近の傾向をより正しく理解できるようになると考えている．さらにその知見は，解析回避に対するより洗練された対策をサンドボックスなどのセキュリティシステムに組み込むことに貢献しうると考えている．

本研究ではまず，マルウェアが RDTSC 命令を実行している部分の周辺の命令列を抽出し，命令列の特徴に従ってグループに分類した．その上で，各グループに属するコードが RDTSC 命令を利用する理由や目的を著者が推定した．その結果，RDTSC 命令を用いて所要サイクル数が計測されている処理は多岐に渡ること，CPUID 命令の実行はそれらの多数派ではないこと，所要サイクル数を計測する以外の目的で RDTSC 命令が実行されている可能性があることなどがわかった．

性能情報を利用する解析回避処理には，RDTSC 命令ではない手段を用いるものも多い．例えば，時間情報を取得するライブラリ関数やシステムコールを用いるものもあり，Windows の環境では `GetSystemTimeAsFileTime`，`GetTickCount`，`QueryPerformanceCounter` などの関数をその用途に用いることができる．しかし，本研究では，分析対象の範囲を絞って，得られる知見をできるだけ具体的なものにするために，それらは扱わない．RDTSC 命令に関する実態についてある程度明らかにした後に，それらの処理にも分析対象を広げることが望ましいと考えている．

2. RDTSC 命令を用いた解析回避への対策

RDTSC 命令による解析回避を防ぐための 1 つの効果的な方法は，RDTSC 命令が返す TSC の値を改変して偽の時間経過をマルウェアに伝えることである．しかし，マルウェアが RDTSC 命令を実行した理由や目的がわからなければ，何が適切な改変かを判断できない．それについて，以下で例を用いて説明する．

RDTSC 命令を用いた解析回避処理のコード例を**図 1** に示す．`RDTSC()` と `CPUID()` はそれぞれ，RDTSC 命令と CPUID 命令を実行するコードを表す．`SLEEP()` は引数の秒数だけスリープするコードを表す．変数 `cpu_freq` には CPU の周波数の値が入っていると仮定する．

```
BOOL detect_vm()
{
  a = RDTSC();
  CPUID();
  b = RDTSC();
  return (b - a > 1000);
}

BOOL detect_sandbox()
{
  a = RDTSC();
  SLEEP(3600); /* 1 hour */
  b = RDTSC();
  /* Did I sleep for over 50 min? */
  return (b - a < cpu_freq * 60 * 50);
}

void busy_sleep(int sleep_duration)
{
  a = RDTSC();
  do {
    b = RDTSC();
  } (b - a > cpu_freq * sleep_duration);
}
```

図 1 RDTSC 命令を用いた解析回避処理のコード例

関数 `detect_vm` は，仮想マシン上で CPUID 命令の実行に多くのサイクルが消費されることが多いことを利用して，仮想マシンの存在を検査する．仮想マシンを検出させないための 1 つの対策は，2 回の RDTSC 命令の実行で，互いに近い TSC の値を返すようにすることである．

関数 `detect_sandbox` は，一部のサンドボックスがスリープをスキップすることを利用して，サンドボックスの存在を検査する．サンドボックスを検出させないための 1 つの対策は，取得される 2 つの TSC の差を，スリープ時間に相当するサイクル数に近くすることである．

関数 `busy_sleep` は，ビジーループを用いて事実上のスリープ処理を実行する．スリープのためのシステム関数を呼び出すと，解析回避処理を実行したとサンドボックスがみなすことがあるが，ビジーループを使えばそれを避けることができる．この処理に対しては，RDTSC 命令の実行で取得される TSC の値をナイーブに改変すると，実行が無限ループに陥ることがある．

これらの例からわかることは，TSC の値をどのように偽るべきかは，RDTSC 命令を実行する目的に依存するということである．RDTSC 命令が返す値を常に同じにしたり，命令の実行のたびにその値に定数を加えたりする単純な方法では，解析が失敗することがある．

3. 方法

マルウェアが RDTSC 命令をどのように利用しているかを分析した．分析の方法を以下で述べる．

分析対象としたマルウェア検体は，アンチウィルスベン

表 1 分類結果

グループ	コードの特徴	コード断片数	検体数	マルウェアの種類数	種類不明の検体数
1	STOS 命令と LOOP 命令によるバッファ間データコピー	363	363	1	5
2	TSC と `GetTickCount()` の XOR 演算	144	48	1	1
3	`Sleep` 関数の実行	97	97	11	4
4	実質的な NOP 命令の列	67	67	1	1
5	10 万回のカウンタデクリメント	39	39	9	0
6	ほぼ連続する RDTSC 命令の結果の差分を定数と比較	26	25	7	7
7	RDTSC 命令の直前に CPUID 命令を入れて処理を実行	8	8	6	0
8	TSC の下位 32 ビットと `GetTickCount()` の XOR 演算	8	8	1	1
9	`GetTickCount()` のループの実行	6	6	4	0
10	XCHG 命令のみの実行	6	6	1	0
11	`QueryPerformanceCounter` 関数を呼び出すループの実行	5	5	3	2
12	`QueryPerformanceCounter` 関数を呼び出すループを実行する関数の実行	4	4	2	2
13	関数の先頭と末尾での RDTSC 命令の実行	4	2	0	2
	その他	13			

ダ等に対してマルウェア検体を提供しているサービスの Web サイトから著者がダウンロードした 110,042 ファイルである．これらはすべて Windows 用の 32 ビットの実行可能バイナリである．まず，これらのファイルに file コマンドを適用して，ファイルがパックされているかどうかを調べた．UPX でパックされていると file コマンドが報告したファイルは，UPX コマンドによってアンパックした．他のパッカー（PECompact2 や petite）でパックされていると file コマンドが報告したファイルは，分析対象から外した．

次に，分析対象のファイルを objdump コマンドによって逆アセンブルして命令列を得た．中身のデータが壊れているなどの理由で逆アセンブルができないファイルは分析対象から外した．さらに，得られた命令列から，50 命令以内の範囲に出現する RDTSC 命令の組を探索し，各組の周辺のコード断片を抽出した．1 つの検体から複数のコード断片が抽出されうる．コード断片が抽出された検体の数は 938 である．

RDTSC 命令が出現するすべての部分を抽出せず，狭い範囲で複数回 RDTSC 命令が出現する部分だけを抽出している理由は，解析回避のための RDTSC 命令の利用に焦点を絞って分析するためである．典型的な解析回避のためのコードでは，サイクル数を計測する処理の前後で RDTSC 命令が実行される．ただし，RDTSC 命令の実行を別関数に分けたなどの理由により，そのような処理の前後で RDTSC 命令が実行されるにもかかわらず，命令列の文面上では RDTSC 命令を単独で実行するようなコードも存在する．そのようなコードも分析することが望ましいが，作業コストの理由から，本研究では分析対象から外した．

抽出したコード断片には，本来コード部分ではない部分を逆アセンブルして得られたと思われる命令列（ゴミ命令列）も含まれている．例えば，file コマンドが認識しないパッカーでファイルがパックされている場合には，objdump が暗号データを逆アセンブルしている可能性がある．そこで，単純なヒューリスティクスによって，ゴミ命令列からなると予想されるコード断片を分析対象から外した．具体的には，以下の命令を含むコード断片を分析対象から外した．

- 無効命令（objdump の出力では `(bad)` や `data16` など）
- 特権命令（HLT 命令や IN 命令など）
- 稀にしか実行されないと判断した命令（AAA 命令や AAS 命令など）
- 取得した TSC の値をアドレスとして使用している命令（RDTSC 命令直後の `mov [eax], ecx` など）

この段階で，20 個以上という多くのコード断片が分析対象になっている 4 検体については，一括処理を避け個別に分析することが好ましいと判断し，それらに由来するコード断片すべてを分析対象から除外した．一連の作業の結果，1204 個のコード断片を得た．

得られたコード断片を特徴に従って分類した．自動的な分類を実現するために，著者が各コード断片を読み，該当する特徴を持つコード断片を検出できるパターンマッチ関数の集合を記述した．抽出したコード断片はどれも，ただ 1 つのパターンマッチ関数にマッチし，22 のグループに分類された．

4. 結果

3 章で述べた方法によってコード断片を抽出して，グループに分類した結果を**表 1** に示す．表中の「マルウェアの種類数」の列は，各検体のマルウェア名を VirusTotal での Microsoft 製品による判定結果にしたがって決定し，それに従って計算した数を示すものである．VirusTotal で情報が提供されていない検体や，Microsoft 製品がマルウェアではないと判定した検体も存在した．それらの検体の数は

```
            rdtsc
            push    eax
            mov     bh, [esi]
            mov     [edi], bh
            inc     edi

loc_4012E9:
            inc     esi
            inc     esi
            push    eax
            mov     al, [esi]
            stosb
            add     [edi-1], bl
            pop     eax
            loop    loc_4012E9

            rdtsc
            pop     edx
            sub     eax, edx
            pop     edx
            push    0
            push    has
            call    ds:acmStreamClose
            sub     edx, eax
```

図 2 STOS 命令と LOOP 命令によるバッファ間データコピー

```
    mov     esi, ds:GetTickCount

    call    esi ; GetTickCount
    mov     [esp+14h+var_10], eax
    rdtsc
    xor     eax, edx
    xor     [esp+14h+var_10], eax

    call    esi ; GetTickCount
    mov     [esp+14h+var_C], eax
    rdtsc
    xor     eax, edx
    xor     [esp+14h+var_C], eax

    call    esi ; GetTickCount
    mov     [esp+14h+var_8], eax
    rdtsc
    xor     eax, edx
    xor     [esp+14h+var_8], eax

    call    esi ; GetTickCount
    mov     [esp+14h+var_4], eax
    rdtsc
    xor     eax, edx
    xor     [esp+14h+var_4], eax
```

図 3 TSC と `GetTickCount()` の XOR 演算

表中の「種類不明の検体数」の列に示されている．「マルウェアの種類数」の列の数には，「種類不明という種類」の数は含まれていない．

各グループのコード断片数や検体数には大きなばらつきがある．100 個以上のコード断片を含むグループがある一方で，多くのグループには 10 個以下のコード断片しか分類されていない．以降ではコード断片数の順に各グループの特徴を述べる．以降で示すコード断片は objdump ではなく IDA Pro による出力を整形したものである．

グループ 1 のコード断片では，STOS 命令と LOOP 命令によるメモリバッファ間コピーの前後で TSC を取得している．このグループのコード断片の例を**図 2** に示す．このコードでは，取得した 2 つの TSC の差（すなわちコピーに要したサイクル数）を計算している．しかしそれを何にも使わず捨てている．このグループのコードでは RDTSC 命令を実行する必要性は希薄である．差を利用するコードが無い理由としては，コードの中途半端な改変や，コンパイル時フラグとマクロの相互作用などにより，元々は存在したその部分だけが省かれたというものを 1 つの可能性として検討している．

グループ 2 のコード断片では，`GetTickCount` 関数の返り値，RDTSC 命令で得た TSC の上位 32 ビット，下位 32 ビットの 3 つに対して XOR 演算を適用している．このグループのコード断片の例を**図 3** に示す．`GetTickCount` は Windows API の関数であり，起動後の経過時間をミリ秒単位で返す．`GetTickCount` 関数と RDTSC 命令の組を 4 回実行し，それらに XOR 演算を適用して得られた 4 つの値をスタックに格納している．図 3 に続く命令列では，それら 4 つの値にさらに XOR 演算と左シフト演算を適用して単一の 32 ビットの値を計算し，それを関数の返り値として返している．この値は処理に要するサイクル数を表すものではないため，このコードは所要時間計測以外の目的で RDTSC 命令を実行していると考えられる．そのような目的の例としては乱数の種が考えられるが，目的の特定には至っていない．今回の分析では，所要時間計測が目的ではないにもかかわらず狭い範囲で RDTSC 命令を複数回実行するコード断片が対象に含まれてしまうことがあるが，このグループはそれに該当する．

グループ 3 のコード断片の例を**図 4** に示す．このグループのコードでは `Sleep` 関数による 500 ミリ秒のスリープに要するサイクル数を計測している．図 4 の命令列に続く部分では，そのサイクル数を 50 万で割った値を関数の返り値として返している．500 ミリ秒は 50 万マイクロ秒であるため，割り算の結果の値は 1 マイクロ秒あたりのサイクル数の増加幅，すなわち，MHz で表した CPU 周波数とほぼ等しい．返り値を受け取る側の関数では，直後に別の関数を呼び出す．その関数の第 1 引数は`"%f MHz"`，第 2 引数はその返り値，第 3 引数が 0 である．第 1 引数の文字列は `printf` 関数などで用いられる書式に沿っており，返り値を`"%f"`の部分に埋め込んだ文字列を生成していると考えられる．一般に，スリープの前後での TSC の差分の取得は，よく知られた解析回避処理である．その差が本来観測されるはずだった値に近いかどうかを検査することにより，サンドボックスによるスリープのスキップを検出することができる [18]．ただし，図 4 に示したコードに関する限りは，その目的ではなく，CPU 周波数を求める目的で TSC を取得していると推測している．

グループ 4 のコード断片は，RDTSC 命令の組の周辺に，同一のレジスタオペランドを持つ MOV 命令が多く出現するというパターンにマッチするものである．このグループのコード断片の例を**図 5** に示す．このコード断片では，最

```
    rdtsc
    mov     [ebp+var_4], eax
    mov     [ebp+var_8], edx
    push    1F4h ; dwMilliseconds
    call    Sleep ; Sleep(500)
    rdtsc
    sub     eax, [ebp+var_4]
    sbb     edx, [ebp+var_8]
```

図 4 Sleep 関数の実行

```
            mov     ebx, 3AD7D97Bh

loc_46388C:
            mov     eax, eax
            dec     ebx
            jnz     short loc_46388C

            rdtsc
            nop
            mov     eax, eax

loc_463896:
            rdtsc
            sub     eax, eax
            ja      short loc_463896

            xchg    edx, edx
            mov     esi, esi
            mov     esi, esi
            mov     ebx, ebx
            nop
            mov     esi, esi
            nop
```

図 5 実質的な NOP 命令の列

```
            rdtsc
            mov     ecx, 186A0h ; 100000

loc_44E310:
            dec     ecx
            jnz     short loc_44E310

            mov     ebx, eax
            rdtsc
            sub     eax, ebx
            mov     [ebp+var_8], eax
```

図 6 10 万回のカウンタデクリメント

初の RDTSC 命令で取得した TSC を 2 つ目の RDTSC 命令で上書きしている．また，2 つ目の RDTSC 命令で取得した TSC も使わないまま捨てている．このコード中のほとんどの命令は実質的には NOP 命令である．2 つの RDTSC 命令も，実行に意味がないため，難読化のために導入されていると推測する．

グループ 5 のコード断片では，カウンタをデクリメントしながら 10 万回回るだけの単純なループの実行に要するサイクル数を計測している．このグループのコード断片の例を**図 6** に示す．このグループのコードでは，2 つ目の RDTSC 命令の実行直後に SUB 命令で TSC の差を取り，ループの実行に要したサイクル数を取得している．

グループ 6 のコード断片の例を**図 7** に示す．このグループのコードは，1 つ目の RDTSC 命令の実行の後に TSC の

```
            rdtsc
            mov     [ebp+var_8], eax
            rdtsc
            mov     [ebp+var_C], eax
            ...
            mov     eax, [ebp+var_C]
            sub     eax, [ebp+var_8]
            cmp     eax, 64h ; 100
            jbe     short loc_406272
            mov     [ebp+var_10], 1
            jmp     short loc_406279
loc_406272:
            mov     [ebp+var_10], 0
loc_406279:
            mov     al, byte ptr [ebp+var_10]
            ...
            retn
```

```
            rdtsc
            mov     [ebp+var_14], eax
            rdtsc
            mov     [ebp+var_10], eax
            mov     eax, [ebp+var_10]
            sub     eax, [ebp+var_14]
            cmp     eax, 1F4h ; 500
            jbe     short loc_40A433
            ...
            call    ds:ExitProcess
loc_40A433:
            ...
```

図 7 ほぼ連続する RDTSC 命令の結果の差分を定数と比較

待避だけを実行し，2 つ目の RDTSC 命令の実行の後に 2 つの TSC の差を定数と比較するものである．図 7 上のコードでは，100 との比較の結果に応じて分岐し，100 より大きい場合には 1 を，そうでない場合には 0 を関数の返り値として返している．図 7 下のコードでは，500 との比較の結果に応じて分岐し，500 より大きい場合には ExitProcess 関数を呼び出している．このグループのコード断片は，典型的な解析回避処理としてよく知られているものである．

グループ 7 のコード断片の例を**図 8** に示す．このグループのコードでは RDTSC 命令の直前で，EAX，EBX，ECX，EDX レジスタの値を全て 0 にした上で CPUID 命令を実行している．RDTSC 命令によって計測している処理は様々であるが，図の例では，timeGetTime 関数を用いて事実上の 1000 ミリ秒のスリープを実行している．

CPUID 命令の実行よりも 1 秒のスリープにはるかに長い時間がかかるため，CPUID 命令の実行による TSC の変化は，ほとんど見えないはずである．よって，このコード断片で CPUID 命令を呼び出している目的は，その所要時間を計測するためではなく，RDTSC 命令が out-of-order 実行されることを防ぐためだと推測している．RDTSC 命令の直前での CPUID 命令の実行は，Intel 社による文書 [7] で推奨されており，既存研究 [11] でも利用されている．

CPUID 命令に要するサイクル数の計測は，仮想マシン検出のための効果的な処理としてよく知られている．しかし，今回の分析では，それを試みていると思われるコード断片は少ししか検出されなかった．これは，その処理を用いるマルウェアは必ずしも多くないという可能性を示唆す

```
            xor     eax, eax
            xor     ebx, ebx
            xor     ecx, ecx
            xor     edx, edx
            cpuid
            rdtsc
            mov     [ebp+var_8], eax

loc_402A95:
            call    edi ; timeGetTime
            sub     eax, esi
            cmp     eax, 3E8h
            jle     short loc_402A95

            xor     eax, eax
            xor     ebx, ebx
            xor     ecx, ecx
            xor     edx, edx
            cpuid
            rdtsc
            mov     [ebp+var_4], eax
            mov     edx, [ebp+var_8]
            mov     ecx, [ebp+var_4]
            sub     ecx, edx
```

図 8 RDTSC 命令の直前に CPUID 命令を入れて処理を実行

```
    rdtsc
    xor     dword_40A000, eax
    mov     [esi+58h], ecx
    call    ebx ; GetTickCount
    mov     ecx, dword_40A000
    xor     ecx, eax
    mov     eax, 80008001h
    mul     ecx
    shr     edx, 0Fh
    add     edx, ecx
    mov     [edi], dx
    rdtsc
    xor     ecx, eax
    mov     dword_40A000, ecx
    call    ebx ; GetTickCount
    mov     ecx, dword_40A000
    xor     ecx, eax
```

図 9 TSC の下位 32 ビットと `GetTickCount()` の XOR 演算

```
            call    GetTickCount
            mov     ebx, eax

loc_4AC19D:
            call    GetTickCount
            cmp     ebx, eax
            jz      short loc_4AC19D

            rdtsc
            mov     [ebp+var_4], edx
            mov     [ebp+var_8], eax

loc_4AC1AE:
            call    GetTickCount
            lea     edx, [ebx+3E8h]
            cmp     eax, edx
            jb      short loc_4AC1AE

            rdtsc
            mov     [ebp+var_C], edx
            mov     [ebp+var_10], eax
```

図 10 `GetTickCount()` のループの実行

```
    rdtsc
    xchg    eax, ecx
    rdtsc
    sub     eax, ecx
```

図 11 XCHG 命令のみの実行

```
    push    [ebp+arg_8]
    mov     ebx, [ebp+arg_4]
    rdtsc
    mov     esi, eax
    mov     edi, edx
    ; Eventually calls QueryPerformanceCounter
    call    ebx
    rdtsc
```

図 12 `QueryPerformanceCounter` 関数を呼び出すループを実行する関数の実行

る．今後，検出が少なかった理由などについて，さらなる調査を行う必要がある．

グループ 8 のコード断片の例を**図 9** に示す．このグループのコードでは，グループ 2 のコードと同じく，TSC の値と `GetTickCount` 関数の返り値に対して XOR 演算を適用している．このグループのコードにおける RDTSC 命令は，グループ 2 のコードにおけるそれと同様に，所要サイクル数を計測するためのものではないと考えられる．

グループ 9 のコード断片は，`getTickCount` 関数のループを実行するというパターンにマッチするものである．このグループのコード断片の例を**図 10** に示す．このコード断片は `getTickCount` 関数の返り値が 1000 増加するまでループを回ることにより，事実上の 1000 ミリ秒のスリープを実現している．

グループ 10 のコード断片は，2 つの RDTSC 命令の間に，TSC の下位 32 ビットを別のレジスタにセットする XCHG 命令が 1 つだけあり，2 つ目の RDTSC 命令の直後に TSC の下位 32 ビットの差分を計算するというパターンにマッチするものである．このグループのコード断片の例を**図 11** に示す．

グループ 11，12 のコードでは，それぞれ，`QueryPerformanceCounter` 関数を呼び出すループの実行に要するサイクル数と，`QueryPerformanceCounter` 関数を呼び出すループを実行する関数の実行に要するサイクル数を計測している．グループ 12 のコード断片の例を**図 12** に示す．このコード断片で呼び出している関数では，`QueryPerformanceCounter` 関数が返す高分解能性能カウンタの値が所定の数だけ増加するまで回るループを実行する．RDTSC 命令によって計測したサイクル数は関数の返り値として返される．

グループ 13 のコードでは関数の先頭と末尾で RDTSC 命令を実行する．グループ 13 のコード断片の例を**図 13** に示す．このコード断片の関数では，関数の実行に要したサイクル数が，所定のメモリアドレス上の値に足される．2 つの RDTSC 命令の間には，2 つの CALL 命令を含む 32 命令が存在する．

```
sub_45EE10:
            push    eax
            push    edx
            rdtsc
            sub     ds:dword_50EF50, eax
            ...
            rdtsc
            add     ds:dword_50EF50, eax
            retn
```

図 13 関数の先頭と末尾での RDTSC 命令の実行

5. 議論

5.1 離れた RDTSC 命令によるサイクル数の計測

今回の分析では，文面上互いに近い場所にある RDTSC 命令の組だけを対象とした．しかし現実には，そうではない RDTSC 命令による解析回避処理は多く存在する．たとえば，RDTSC 命令によって TSC の値を取得して返すだけの関数を導入し，サイクル数を計測したい処理の前後でその関数を呼び出すものが考えられる．今回用いた方法を今後，制御フロー解析や動的解析との組み合わせなどによって拡張し，文面上互いに離れているが時間的には隣接して実行される RDTSC 命令の組も認識できるようにすることが望ましいと考えている．

5.2 分析対象から外れた検体

今回の分析では，アンパックできない検体や，RDTSC 命令の周辺に無効命令などの特殊な命令が出現する検体を対象から外した，しかし実際には，それらの検体も RDTSC 命令を用いた解析回避処理を実行している可能性がある．本研究では，分析作業のコストを下げるために，有効である可能性が高い命令列を単純な処理によって得られる検体のみを分析対象とした．長期的には，洗練されたアンパックやコード分析を導入することにより，より広い範囲の検体を分析対象に含める必要があると考えている．

5.3 セキュリティシステムへの適用

本研究で調査したような，各マルウェアによる RDTSC 命令の利用方法についての情報を，現実のセキュリティシステムで有効に活用することは可能であると考えている．まず，マルウェア解析用のサンドボックスがそのような情報を収集し，それらをシグネチャとして表示できるようにすることが考えられる．既存のサンドボックスのいくつかは，マルウェアのプログラムファイルや挙動から解析回避処理の兆候を検出して，シグネチャとしてユーザに報告する．例えば，Cuckoo Sandbox には，「長いスリープの実行」，「ディスクサイズの検査」，「VMware の存在を示唆するファイルへのアクセス」などの個々の解析回避処理に対してシグネチャが定義されている．今後，それらのサンドボックスを拡張し，「スリープのサイクル数の計測」や「CPUID 命令のサイクル数の計測」などの，RDTSC 命令の実行に関するシグネチャも報告できるようにすることは興味深いテーマである．

また，サンドボックスをさらに拡張し，RDTSC 命令を用いた解析回避処理への対策を組み込むことが考えられる．我々の知る限り，各サンドボックスによるそのような対策は，存在しないか，ほとんど明らかにされていない．例えば Cuckoo Sandbox では，RDTSC 命令の実行に対しては特に何も行わない．多くのクローズドソースのサンドボックスでは，当該ベンダが，時間やサイクル数の情報を用いた解析回避処理に対する対策を組み込んでいると述べているが，詳細が不明確であることが多い．例えばそれらのサンドボックスが TSC の値を改変するのかしないのか，改変するならどういうアルゴリズムで改変するのかについては，ほとんど公開されていない．対策の実現方法について，技術をさらに洗練させていくとともに，オープンな形で知見を蓄積していくことが必要であると考える．

さらに，RDTSC 命令を用いた解析回避処理がマルウェアの種類ごとに大きく異なる場合には，その情報をマルウェアの検知や分類に役立てられる可能性がある．少なくとも本研究で分析した限りでは，マルウェアの種類と解析回避処理の特徴の間には強い相関があり，検知や分類への応用を考えることには大きな意味があると考えている．

6. 関連研究

特定の処理にかかる時間やサイクル数の計測によってサンドボックスや仮想マシンを検出する方法については多くの研究がある [2, 3, 6, 11–13, 16–19]．時間に関する解析回避処理や，他の様々な解析回避処理の実態を明らかにする研究も存在する [8, 9]．しかし，どの研究においても，マルウェアが具体的に何を実現するためにどのように RDTSC 命令を利用しているかの実態については，明らかにされていない．本研究はそれを明らかにするものである．

シンボリック実行を用いたマルウェア解析システムが複数提案されている．例えば，angr [15] や KLEE [4] がある．Angr では，与えられた初期状態から与えられた終了状態に至るための入力の条件や例を，シンボリック実行により解析，発見する機能を有している．例えば，あるプログラム地点から別のあるプログラム地点に至るための入力の条件を得ることができる．このようなシステムを用いると，時間やサイクル数の情報を用いた解析回避処理を通過するためにそれらの値が満たすべき条件や通過できる値の例を自動的に求められることがある．実際，本稿で示したコードには，シンボリック実行に基づく TSC の値の自動的な改変により，その効果を無くせるものも存在する．

このようなツールを使うには，通常，初期状態や終了状態を指定する必要があることに注意が必要である．マルウェア解析においては，それらをどう設定するかは自明で

はなく，ましてやサンドボックスに自動的に設定させることは極めて困難である．また，そのようなシステムを用いて入力の例を得たとしても，その例がなぜ良いのかは依然として不明なままであるという問題がある．マルウェア解析の重要な目的はマルウェアの挙動や意図を明らかにすることである．よって，マルウェアに様々な挙動を実行させるための入力の条件や例を得る技術とともに，マルウェアの意図を推定する技術も併用する必要がある．本研究は後者の技術を開発するための知見を提供するものである．

7. まとめと今後の課題

マルウェアが実行する RDTSC 命令の組の周辺にある命令列を抽出し，マルウェアがどういう目的や方法で RDTSC 命令を利用しているかを分析した．マルウェアが所要サイクル数を計測する処理には，バッファ間データコピー，単純なループ，スリープなどがあり，多様であることがわかった．また，おそらく難読化のために意味なく RDTSC 命令を実行したり，複数の TSC の値に対して減算ではなく XOR 演算やシフト演算を適用したりすることがあることもわかった．RDTSC 命令による解析回避処理に対してセキュリティシステムが適切な対策を実行するには，RDTSC 命令を実行する「意図」を正しく判断する必要があると考えるが，本研究はそのための第一歩である．

今後の課題を以下に述べる．第一に，抽出したコード断片には，RDTSC 命令を実行する目的や所要サイクル数の利用方法を完全に特定しきれていないものがあるため，それらを特定する必要がある．例えば本研究ではどんな処理の所要サイクル数が計測されているかを中心に分析を行ったが，マルウェアがそのサイクル数情報をどう利用して，どう解析を回避するのかまでは，十分に解明していない．今後それらについての調査が必要である．第二に，分析方法を拡張して，文面上だけではなく時間的に連続して実行されている RDTSC 命令の組も分析する必要がある．今回は静的解析のみによってコードを分析したが，今後，動的解析も組み合わせることが必要である．第三に，本研究での分析による知見に基づいて，様々な目的での RDTSC 命令の実行に対して賢く対処するための方法を確立することが必要である．

謝辞 本研究に対して，情報セキュリティ大学院大学の窪優司氏，筑波大学の小池悠生氏，株式会社富士通研究所の小久保博崇氏から有益な意見をいただいた．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] Al-Khaser, `https://github.com/LordNoteworthy/al-khaser/`.

[2] 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 USA 2012 (2012).

[3] Brengel, M., Backes, M. and Rossow, C.: Detecting Hardware-Assisted Virtualization, *Proceedings of the 13th International Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, pp. 207–227 (2016).

[4] Cadar, C., Dunbar, D. and Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (2008).

[5] Forcepoint Security Labs Blog: Locky Returned With A New Anti-VM Trick, `https://blogs.forcepoint.com/security-labs/locky-returned-new-anti-vm-trick` (2016).

[6] Franklin, J., Luk, M., McCune, J. M., Seshadri, A., Perrig, A. and van Doorn, L.: Towards Sound Detection of Virtual Machines, *Botnet Detection*, Advances in Information Security, Vol. 36, pp. 89–116 (2008).

[7] Intel: How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures, `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf` (2010).

[8] Oyama, Y.: Trends of anti-analysis operations of malwares observed in API call logs, *Journal of Computer Virology and Hacking Techniques* (2017).

[9] Oyama, Y.: Investigation of the Diverse Sleep Behavior of Malware, *Journal of Information Processing*, Vol. 26 (2018).

[10] Pafish (Paranoid Fish), `https://github.com/a0rtega/pafish/`.

[11] Pék, G., Bencsáth, B. and Buttyán, L.: nEther: In-guest Detection of Out-of-the-guest Malware Analyzers, *Proceedings of the 4th European Workshop on System Security* (2011).

[12] Raffetseder, T., Kruegel, C. and Kirda, E.: Detecting System Emulators, *Proceedings of the 10th Information Security Conference*, pp. 1–18 (2007).

[13] Rutkowska, J. and Tereshkin, A.: IsGameOver() Anyone?, Black Hat USA 2007 (2007).

[14] Shi, H., Mirkovic, J. and Alwabel, A.: Handling Anti-Virtual Machine Techniques in Malicious Software, *ACM Transactions on Privacy and Security*, Vol. 21, No. 1 (2017).

[15] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C. and Vigna, G.: (State of) The Art of War: Offensive Techniques in Binary Analysis, *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, pp. 138–157 (2016).

[16] Vasudevan, A. and Yerraballi, R.: Cobra: Fine-grained Malware Analysis using Stealth Localized-executions, *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006).

[17] 宮本久仁男，田中英彦：特徴データベースを用いない効率的な仮想マシンモニタ検出方式の提案，情報処理学会論文誌，Vol. 52, No. 9, pp. 2602–2612 (2011).

[18] 大山恵弘：動的マルウェア解析においてスリープ時間を短縮する方式，コンピュータセキュリティシンポジウム 2017 論文集，pp. 487–494 (2017).

[19] 大山恵弘：Raspberry Pi 環境におけるステルス性の高い仮想マシン検出，情報処理学会研究報告コンピュータセキュリティ（CSEC），Vol. 2018-CSEC-81 (2018).