SleepHop: 動的バイナリ計装によるマルウェアのタイミング攻撃の無効化

鈴木 克弥[1,a)]　大山 恵弘[1]

概要：マルウェアの多くは，セキュリティシステムによる解析を回避しようとする処理（解析回避処理）を行う．McAfee Labs の 2017 年の脅威レポートによると，解析回避処理のうち約 23.3%がサンドボックスを回避する技術を使用している．サンドボックス回避技術の 1 つに，CPU サイクル数やスリープを用いたタイミング攻撃が存在するが，既存のオープンソースのサンドボックスの多くは対策をしていない．そこで本研究では，動的バイナリ計装（DBI）を使用してタイミング攻撃を回避するシステム SleepHop を提案し，その有効性を実験により検証する．具体的には，CPU サイクル数を用いたタイミング攻撃には偽装したサイクル数を返すようにし，スリープを用いたタイミング攻撃にはスリープをスキップし，経過時間の偽装をするという方針を取る．SleepHop を DBI システムのプラグインとして実装し，タイミング攻撃を実装した仮想的なマルウェアと解析回避処理の PoC プログラムである Pafish をターゲットとして評価実験を行った．実験の結果，タイミング攻撃のほとんどを回避することが出来た．

キーワード：動的バイナリ計装，サンドボックス，マルウェア

SleepHop: Prevention of Malware Timing Attacks Using Dynamic Binary Instrumentation

Katsuya Suzuki[1,a)]　Yoshihiro Oyama[1]

Abstract: Many malware execute operations for evading analysis by security systems (analysis evasion operations). According to McAfee Labs' 2017 threat report, about 23.3% of analysis evasion operations use anti-sandbox techniques. One of the anti-sandbox techniques is timing attack using the number of CPU cycles or sleep, but most of the existing open source sandboxes do not provide any countermeasures. In this paper, we propose SleepHop, a system that prevents timing attacks using a dynamic binary instrumentation (DBI), and verify its effectiveness through experiments. Specifically, SleepHop returns a fake value for timing attacks using the number of CPU cycles, and skips sleep and fakes the elapsed time for timing attacks using sleep. We implemented SleepHop as a plug-in for the DBI system, and conducted evaluation experiments targeting a hypothetical malware that implements timing attacks and Pafish, a PoC program for analysis evasion operations. As a result of the experiments, SleepHop was able to prevent most of the timing attacks.

Keywords: dynamic binary instrumentation, sandbox, malware

1. はじめに

マルウェアの多くはセキュリティシステムによる解析を回避しようとする処理（解析回避処理）を実行することが知られている [6,10,12,14,15,19]．これによって，マルウェアは自身の挙動を隠蔽することが可能になる．解析回避処理には，実行バイナリに難読化を施したり，実行環境でセキュリティツールが使用されていることを検知したら実行を終了するなど，様々な種類が存在する．これによ

[1] 筑波大学
University of Tsukuba
[a)] suzuki@syssec.cs.tsukuba.ac.jp

り，マルウェアの解析に時間がかかったり，マルウェアが実行を終了してしまって解析ができない場合がある．マルウェアを解析するためには，こうした解析回避処理を無効化して，マルウェア作者が意図した挙動をあえて実行させる必要がある．このような解析回避処理の 1 つに，サンドボックスの回避がある．McAfee Labs による 2017 年の脅威レポート [6] によると，マルウェアでよく利用される解析回避処理のうち，約 23.3%がサンドボックスを回避する技術を使用している．デバッガや仮想マシンモニタなど，マルウェアが回避したい対象がいくつかある中で，サンドボックスは大きな割合を占めている．いくつか存在するサンドボックス回避手法の分類の 1 つに，タイミング攻撃がある．タイミング攻撃とは，主に時間経過を利用するサンドボックスの回避方法である．タイミング攻撃については 2 章で詳しく説明する．本論文で取り上げるタイミング攻撃の 1 つは，スリープを使った攻撃である [16, 17]．マルウェアは，解析システムをタイムアウトさせることなどを目的にスリープする．つまり，スリープを使った攻撃とは，解析システムをタイムアウトさせたり，解析に膨大な時間を掛けさせる攻撃のことである．こうしたタイミング攻撃は現在でも有効なサンドボックスの回避方法であり，Cuckoo Sandbox [3] などの既存のサンドボックスの多くは対策をしていない．そこで本研究では，サンドボックスを回避するマルウェアによる攻撃のうち，CPU サイクル数とスリープを利用したタイミング攻撃を無効化するシステム SleepHop を提案する．SleepHop は，動的バイナリ計装（Dynamic Binary Instrumentation, DBI）の技術を使用し，マルウェアのスリープをスキップしつつ，実行環境がサンドボックスであるとマルウェアに推定されないようにするシステムである．本研究の対象 OS は Windows であり，対象 CPU アーキテクチャは x86 である．また，対象とするマルウェアは Windows マルウェアである．SleepHop の有効性を示すために，タイミング攻撃を実装した仮想的なマルウェアを作成し，これと解析回避処理の PoC プログラムである Pafish [7] をターゲットとして評価実験を行った．実験の結果，仮想的なマルウェアによるサンドボックス回避のほとんどを無効化することができた．本研究の貢献は，これまでに提案された手法を実装し，実験を通してその有効性を評価した点にある．

2. タイミング攻撃

本研究で対象としているマルウェアのタイミング攻撃について説明する．

2.1 CPU サイクル数を用いたサンドボックス検出

何らかの処理に要する CPU サイクル数を計測し，閾値以上であれば仮想環境であると推定する方法がある．例として，プロセッサを識別する x86 命令である CPUID 命令の実行に要する CPU サイクル数を計測する方法が有効であることが知られている [11]．これは，Intel プロセッサによる仮想化支援機構である VT-x が働いている環境において，ゲスト環境から仮想マシンモニタに制御が移ること (VMExit) に時間がかかることを利用している．CPUID 命令を含む，仮想マシンモニタの構成に依存するいくつかの命令を実行する際は必ず VMExit が発生する．そのため，CPUID 命令にかかる CPU サイクル数を計測することで，サンドボックス，ひいては仮想環境を検出することが可能になる．

2.2 スリープによるサンドボックス回避

解析システムにはタイムアウト時間を設けているものが多い．スリープをするマルウェアをそのまま解析システムで実行すると，このタイムアウト時間に到達したり，少なくとも解析に時間がかかる場合が多い．これを利用して，自身の解析を回避するために長時間のスリープをするマルウェアが存在する．マルチスレッドを考慮しない場合の典型的な攻撃方法の例として，Windows API の `Sleep` 関数を呼び出して長時間スリープするという方法がある．この方法を用いることで，短い時間のタイムアウトを設けているサンドボックスによる解析を回避することができる．これに対する防御策として，実際にスリープする時間を極端に短くするという方法がある．例として，Cuckoo Sandbox はスリープ時間を 0.1 秒に変更する．マルウェアがマルチスレッドを使用している場合，このような単純なスリープ時間の短縮は有効ではないことがある．なぜなら，マルチスレッドプログラムにおいては，実行の非決定性により，スリープ短縮後のマルウェアの各処理の実行順序がスリープ短縮前から変わる可能性があるからである．単にスリープを短縮するだけではマルウェアの挙動を意図せず変える可能性がある．このことは，マルウェアの本来の動作を変えないことが望ましいことや，動作変更をマルウェアが検知して利用する可能性があることから，避けるべきである．また，スリープに使用されうる Windows API はいくつか存在するが，それら全てがスリープを主目的としたものではないため，単純にスリープ時間を短縮するだけでは，意図していない挙動までも抑制する可能性がある．

スリープをするために使用される Windows API として本研究が扱うものは以下の 7 つである．

- `NtDelayExecution`
- `Sleep`
- `SetTimer`
- `WaitForSingleObject`
- `IcmpSendEcho`
- `SetWaitableTimer`
- `CreateTimerQueueTimer`

これらは以下の 3 つに分類することができる．

スリープを主目的とする関数 NtDelayExecution, Sleep が該当する．これらは現在のスレッドの実行を中断することが目的の関数である．

タイムアウト時間を指定する関数 WaitForSingleObject，IcmpSendEcho が該当する．WaitForSingleObject はシグナルの受信，IcmpSendEcho は ICMP の Echo Request の送信が主目的である．どちらも引数にタイムアウト時間を指定することができ，これを利用して事実上のスリープを実現する．

タイマ関数 SetTimer，SetWaitableTimer，CreateTimerQueueTimer が該当する．これらは指定時間後にコールバック関数の実行やシグナルの送信をする．これを利用して事実上のスリープを実現する．指定時間後に実行される処理が主目的の関数である．

上記の分類のうち，スリープを主目的とする関数とタイムアウト時間を指定する関数は，それ単体のみでタイミング攻撃として使用することが可能である．一方，タイマ関数は指定時間後に実行される処理を受け取る関数を実行する必要がある．例えば，SetTimer は GetMessage と，SetWaitableTimer と CreateTimerQueueTimer は WaitForSingleObject と組み合わせて使用することで，スリープを実現することが可能になる．

2.3 CPU サイクル数とスリープの組み合わせによるサンドボックス検出

サンドボックスの中には，スリープの時間を変えることでタイミング攻撃に対抗するものが存在する [3,13]．マルウェアはこれを逆手に取って，自身がスリープで指定した時間が実際に経過したかどうかを確認することでサンドボックスを検出することができる．本研究では，このような経過時間とスリープ時間を比較する場合を想定する．具体的には，スリープする関数の前後で CPU サイクル数を取得し，その差分から経過時間を計算する．その後，経過時間とスリープ時間の差を計算し，閾値以上であればサンドボックス上であると推定する．

2.4 実装例

スリープによるサンドボックス回避処理の実装例を**図 1**，**図 2** に示す．

図 1 の関数では，タイムアウト時間を指定する関数に分類される IcmpSendEcho を使用してスリープをしている．17 行目で IcmpSendEcho を実行しており，その前後で CPU サイクル数を取得している．その後，20 行目で経過時間を計算している．この関数は，一連の処理のどれかが失敗するか経過時間が閾値を超えると，実行環境がサンドボックスであると推定する．

```
1  bool sandboxDetectedWithIcmpSendEcho(DWORD sleepTime) {
2      // Preparation for IcmpSendEcho
3          // If it fails, this function returns true.
4      // Preparation for QueryPerformanceCounter
5
6      QueryPerformanceCounter(&sleepStart);
7      IcmpSendEcho(hIcmpFile, NULL, SendData, sizeof(SendData),
           NULL, NULL, ReplySize, sleepTime);
8      QueryPerformanceCounter(&sleepStop);
9
10     double elapsedTime = (double)(sleepStop.QuadPart -
           sleepStart.QuadPart) / freq.QuadPart;
11     if (elapsedTime > THRESHOLD) {
12         return true;
13     }
14
15     // Cleaning up
16
17     return false;
18 }
```

図 1 タイムアウト時間を指定する関数を使用してサンドボックスを検出する関数

```
1  bool sandboxDetectedWithCreateWaitableTimer(LONGLONG
       sleepTime) {
2      // Preparation for SetWaitableTimer
3          // If it fails, this function returns true.
4      // Preparation for QueryPerformanceCounter
5
6      QueryPerformanceCounter(&sleepStart);
7      if (!SetWaitableTimer(hTimer, &dueTime, 0, NULL, NULL,
           false)) {
8          return true;
9      } else if (WaitForSingleObject(hTimer, INFINITE) !=
           WAIT_OBJECT_0) {
10         return true;
11     }
12     QueryPerformanceCounter(&sleepStop);
13
14     double elapsedTime = (double)(sleepStop.QuadPart -
           sleepStart.QuadPart) / freq.QuadPart;
15     if (elapsedTime > THRESHOLD) {
16         return true;
17     }
18
19     // Cleaning up
20
21     return false;
22 }
```

図 2 タイマ関数を使用してサンドボックスを検出する関数

図 2 の関数では，タイマ関数に分類される SetWaitableTimer を使用してスリープしている．11 行目で SetWaitableTimer を実行してタイマをセットし，13 行目で WaitForSingleObject を実行してタイマの満了まで待機している．また，10 行目と 16 行目で CPU サイクル数を取得することで，タイマのセットから満了までの経過時間を計測しようとしている．経過時間の計算やサンドボックス推定基準は 1 つ目の例と同様である．

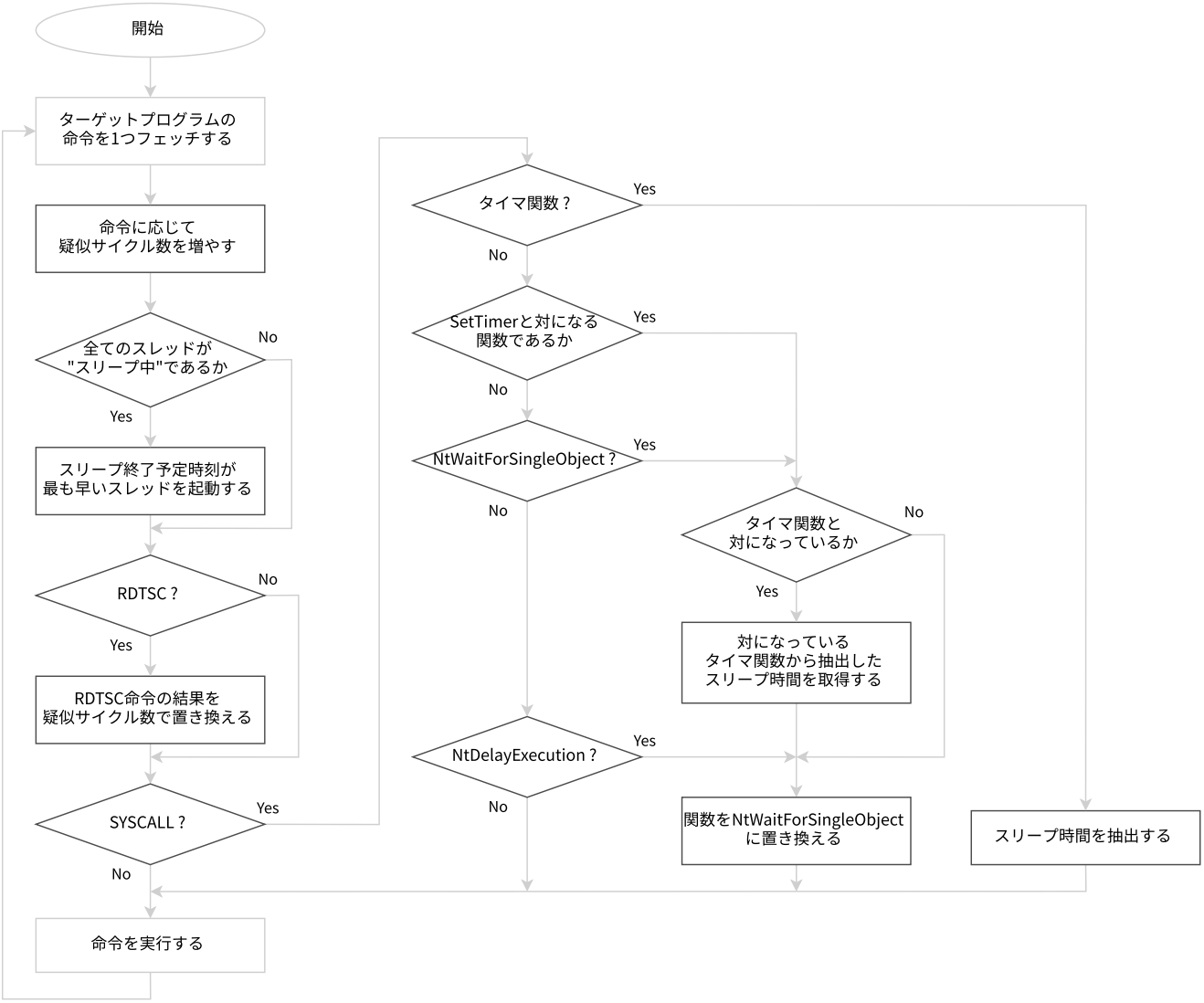

図 3 SleepHop のフローチャート

3. 提案手法

3.1 概要

本研究で提案する SleepHop は DBI の技術を使用する．DBI とは，ターゲットプログラムの解析や挙動の変更を目的とし，実行時のターゲットプログラムに任意のバイナリを注入するシステムである．DBI システムの 1 つである DynamoRIO [4] を使用し，SleepHop の実装を DynamoRIO のプラグインとして実現する．具体的には，DynamoRIO の API を使って RDTSC 命令と Windows API をフックし，RDTSC 命令実行直後のレジスタ値の書き換えや Windows API のシステムコール番号や引数の書き換えを行う．DBI システムは，Intel Pin [5] に代表される他の DBI システムでも代替可能である．本システムの処理の流れの概略を**図 3** に示す．

3.2 CPU サイクル数を用いたサンドボックス検出への対策

CPU サイクル数を用いたサンドボックス検出に対抗するために，SleepHop は疑似的なサイクル数を保持する．擬似的なサイクル数とは，マルウェアに与えるためだけの偽のサイクル数である．この疑似サイクル数は，CPU 命令ごとにあらかじめ実験的に計測された CPU サイクル数 [1] だけ増加させる．そのため，実際にかかった CPU サイクル数は考慮せず，また仮想環境の有無によって値が変わることもない．解析対象プログラムが RDTSC 命令を実行すると，SleepHop は本来の CPU サイクル数の代わりにこの疑似サイクル数を返す．この方式は Blue Pill の研究 [20] で提案された方式と同様である．

3.3 スリープを用いたサンドボックス回避への対策

スリープによるサンドボックス回避に対しては，文献 [23] の研究と同様に実行順序をできるだけ保ったままスリープ

をスキップする方針を取る．

SleepHop は大きく分けて以下の 3 つの操作で構成されている．

スリープ時間の取得　Windows API を呼び出す際の引数からスリープ時間を取得する．実行の非決定性をできるだけ保ちつつスリープをスキップするために，この操作は重要である．

スレッドの監視　SleepHop は，解析対象プログラムの全スレッドの実行状態（実行中かスリープ中）とスリープ情報（開始時刻，終了予定時刻）を監視するテーブル（スレッド監視テーブル）を保持する．

スレッドの起動　スレッド監視テーブルのスレッドが全て“スリープ中”のとき，スリープ終了予定時刻が最も早いスレッドを起動する．それと同時に，スレッド起動時の時刻から起動したスレッドのスリープ終了予定時刻までの時間をスキップ時間とし，スレッド監視テーブルのそれぞれのスリープ終了予定時刻をスキップ時間だけ早める処理を施す．

スリープ時間の修正を容易にするため，解析対象プログラムがスリープを実行すると，SleepHop はそれを `NtWaitForSingleObject` に置き換える．`NtWaitForSingleObject` は，引数で指定したオブジェクトがシグナルを受信するか，指定した時間が経過するまで待機する Windows API である．Windows API の `WaitForSingleObject` は，Windows 内部ではこの `NtWaitForSingleObject` を呼んでいる．スレッド監視テーブルには，スレッドごとに `NtWaitForSingleObject` 用のオブジェクトを登録しておき，スレッドを起動するときは，対応するオブジェクトにシグナルを送信することでスリープからの復帰を実現する．

スリープを `NtWaitForSingleObject` に置き換える際，2 章で分類したカテゴリによって SleepHop の行う処理が異なる．

- スリープを主目的とする関数とタイムアウト時間を指定する関数は，前述の通り関数単体でタイミング攻撃として使用可能なため，スレッド監視テーブルに登録するスリープ情報とシグナル送信用オブジェクトの両方をその関数から取得する．
- タイマ関数に分類されるものは，タイマ関数からスリープ情報を，対になる `GetMessage` などの関数からシグナル送信用オブジェクトを取得する．

3.4 CPU サイクル数とスリープの組み合わせによるサンドボックス検出への対策

最後に，CPU サイクル数とスリープの組み合わせによるサンドボックス検出に対する対策を述べる．上記手法によりスリープをスキップする際，スキップ時間分だけ SleepHop の疑似サイクル数を増加させる．増加させる値は，スキップ時間（秒）に実行環境の CPU の動作周波数（GHz）を掛けたものとする．

表 1　ホストマシンの仕様

CPU	Intel Core i9-9900K (3.60 GHz)
コア数	16
メモリ	32 GB
OS	Ubuntu 20.04.2 LTS
仮想マシンモニタ	QEMU 5.1.0

表 2　ゲストマシンの仕様

仮想 CPU	Intel Core Processor (Skylake) (3.60 GHz)
コア数	2
メモリ	8 GB
OS	Windows 10 Professional 64 bit (21H1)

4. 実験

4.1 概要

提案システムの有効性を評価する実験を行った．実験に先立って，仮想的なマルウェアを作成した．このマルウェアは解析回避処理の PoC プログラムの 1 つである Al-Khaser [2] を基にしている．Al-Khaser が採用している解析回避処理のうち，現在でも有効なもののみを抽出し，さらに 2.3 節で述べた CPU サイクル数とスリープを組み合わせたサンドボックス検出処理を加えた．今回作成した仮想的なマルウェアは以下の 8 つの攻撃手法を採用していることになる．

- `NtDelayExecution` を使用したスリープ
- `Sleep` を使用したスリープ
- `SetTimer` と `GetMessage` を使用したスリープ
- `WaitForSingleObject` を使用したスリープ
- `IcmpSendEcho` を使用したスリープ
- `SetWaitableTimer` と `WaitForSingleObject` を使用したスリープ
- `CreateTimerQueueTimer` と `WaitForSingleObject` を使用したスリープ
- CPUID 命令の実行に要する CPU サイクル数を RDTSC 命令で計測するサンドボックス検出

また，他に代表的な解析回避処理 PoC プログラムとして Pafish [7] がある．Pafish も実験の対象に加える．

実験環境を**表 1**，**表 2** に示す．Ubuntu 20.04.2 LTS のホストマシン上に Windows 10 Professional のゲストマシン環境を作成する．仮想マシンモニタには QEMU 5.1.0 を使用する．DBI システムには DynamoRIO 8.0.0 を使用する．

仮想的なマルウェアと Pafish を以下の 4 つの条件で実行する．

(1) SleepHop を使用せずに実行
(2) SleepHop の疑似サイクル数の機能のみを使用して実行

表 3 仮想的なマルウェアを実行した結果

条件	検出数	実行時間
SleepHop なし	0/8	4809.57 sec
疑似サイクル数のみ … (a)	7/8	4810.03 sec
スリープスキップのみ … (b)	8/8	2.33 sec
(a) + (b)	1/8	4.78 sec

表 4 Pafish を実行した結果

条件	項目 A	項目 B	項目 C
SleepHop なし	OK	detected	OK
疑似サイクル数のみ … (a)	OK	OK	OK
スリープスキップのみ … (b)	detected	detected	detected
(a) + (b)	OK	OK	detected

(3) SleepHop のスリープをスキップする機能のみを使用して実行

(4) SleepHop の全ての機能を使用して実行

SleepHop は疑似サイクル数を実現する機能とスリープをスキップする機能のそれぞれを独立して有効化することが可能なため，これらを網羅するために 4 つの条件を設けている．

4.2 結果

まず，4 つの条件で仮想的なマルウェアを実行した結果を**表 3** に示す．検出数は，実行環境がサンドボックスであると仮想的なマルウェアが推定した数である．実行時間は，同じ条件で仮想的なマルウェアを 10 回実行したときの平均値である．(1) SleepHop を使用せずに実行した場合，仮想的なマルウェアのスリープを使用した攻撃のそれぞれに対して 10 分ずつスリープするため，1 時間 20 分以上の実行時間がかかっている．(2) SleepHop の疑似サイクル数の機能のみを使用して実行した場合，CPUID 命令の実行に要する CPU サイクル数を計測する攻撃のみを回避することが出来た．しかし，実行時間は SleepHop を使用せずに実行したときとほとんど変わらない．(3) SleepHop のスリープをスキップする機能のみを使用して実行した場合，実行時間は大幅に短縮できているが，実際の経過時間とスリープで指定した時間の乖離により，仮想的なマルウェアのすべての攻撃手法によって実行環境がサンドボックス上であると推定されてしまっている．(4) SleepHop の全ての機能を使用して実行した場合，ほとんどのタイミング攻撃を回避しつつ，スリープをスキップさせて実行時間を大幅に短縮できている．唯一サンドボックスであると推定されてしまった攻撃手法については 5 章で議論する．

次に，同様の条件で Pafish を実行した結果を**表 4** に示す．表中の detected は Pafish に実行環境がサンドボックスであると検出された攻撃であり，OK は検出されなかった攻撃である．Pafish が採用している攻撃のうち，本研究が対象としているタイミング攻撃に該当する 3 つのみを表に示している．Pafish は解析システムのタイムアウトを目的としたタイミング攻撃を使用しておらず，スリープをスキップするか否かで実行時間に大幅な差がないため，実行時間は表に記載していない．項目 A の攻撃は，連続で実行した 2 つの RDTSC 命令の返り値の差分値を閾値と比較する．項目 B の攻撃は，2.1 節で述べた攻撃手法である．CPUID 命令の実行に要する CPU サイクル数を RDTSC 命令で計測し，閾値と比較する．項目 C の攻撃は，`Sleep` 関数で 500 ミリ秒スリープしたときの経過時間を `GetTickCount` 関数で計測し，閾値と比較する．スリープがスキップされたかどうかを検査する攻撃である．

(1) SleepHop を使用せずに実行した場合，項目 B の攻撃のが検出される．(2) SleepHop の疑似サイクル数の機能のみを使用して実行した場合，項目 B の攻撃を回避することができ，本研究で対象としている全ての攻撃を通過することができる．(3) SleepHop のスリープをスキップする機能のみを使用して実行した場合，全ての攻撃において Pafish に実行環境がサンドボックスであると推定される．これは，単にスリープをスキップするだけでは Pafish のタイミング攻撃を防ぐことができないことを示している．(4) SleepHop の全ての機能を使用して実行した場合，項目 C の攻撃のみが検出される．Pafish が項目 C の攻撃で使用している `GetTickCount` 関数は OS が起動してからの時間をミリ秒単位で返す関数である．SleepHop は `GetTickCount` 関数で得られる時刻情報の修正は未実装なので，項目 C の攻撃を回避することは本研究の対象外である．

5. 議論

SleepHop によるスリープのスキップをしても 1 つだけ防ぐことできなかった攻撃手法がある．それは，`IcmpSendEcho` を使用した攻撃である．`IcmpSendEcho` の処理を実行するスレッドは `NtDeviceIoControlFile` と `NtWaitForSingleObject` の 2 つの Windows API を呼ぶ．`NtDeviceIoControlFile` を呼んでスレッドプールのワーカスレッドに ICMP の処理を実行させ，自スレッドは `NtWaitForSingleObject` を呼んで ICMP の処理が終了するまで待機する．このとき，待機する時間を `NtWaitForSingleObject` の引数で指定するが，スリープ時間を 1 回の Windows API 呼び出しで待機するのではなく，何らかのアルゴリズムに沿って合計でスリープ時間に達するようにスリープをする挙動になっている．これらの仕様は公開されておらず，現時点ではどんなアルゴリズムで指定された時間待機するのか明らかになっていない．こうした理由から，SleepHop を使用しても `IcmpSendEcho` を使用した攻撃だけは防ぐことができない．

6. 関連研究

Rutkowska らが提案した仮想マシンモニタ Blue Pill [20] では，マルウェアが実行した RDTSC 命令に対して偽の値を返すという方法を取っている．この偽の値は，最後に呼び出された RDTSC 命令の結果にエミュレートされた CPU サイクル数の累積を足したものである．エミュレートされた CPU サイクル数は，実行した CPU 命令に応じて変化する．基本的には本研究と同じ手法を取っているが，エミュレートされた CPU サイクル数を CPU 命令に応じてどのように変化させているかが明記されておらず，また手法の有効性が実験で示されていない．

Shi らの研究 [21] では，解析システムの存在をマルウェアから隠蔽するために，デバッガプラグインに独自の時間カウンタを設ける方法を取っている．マルウェアの解析をする際には，マルウェアが実行した CPU 命令にかかる時間を時間カウンタに加える．マルウェアが時間を取得する命令を実行した際には，CPU サイクル数ではなくその時間カウンタを返すようにしている．これにより，デバッガを使用したステップ実行と通常実行の間での時間の流れの乖離を防ぐことができる．彼らのシステムにおいて偽の時間を計算するための手法は，SleepHop と基本的には同じである．この研究が Windows OS 向けのデバッガである WinDbg のプラグインとして実装されているのに対し，本研究は DBI システムのプラグインとして実装している点や，スリープを使った攻撃手法への対策も行っている点で異なる．

FarFuture [23] はマルチスレッドプログラムのマルウェアであっても挙動を変えずにスリープをスキップする方針を提案している．従来の解析システムにもスリープをスキップする機能は存在していたが，マルチスレッドプログラムにおいては単純なスリープのスキップをすると解析結果が変わってしまう可能性がある．そこで，解析対象プログラムの全スレッドの状態を監視し，全スレッドがスリープ状態であり，かつ最も早くスリープから復帰するスレッドのみを起動することで，実行の非決定性を排除しつつ，スリープをスキップすることができるようになる．これにより，タイミング攻撃を使用するマルウェアが解析システム上で自身の挙動を隠蔽することを防ぐことができた．この研究は Linux 上での動作を想定しているのに対し，本研究は Windows 上でも動作するように実装している．

タイミング攻撃を無効化する技術を実装したツールがいくつか存在する．tiny_tracer [8] は RDTSC 命令を使用したタイミング攻撃を無効化する機能を有した Windows API トレーサである．このトレーサは，DBI ツールの 1 つである Intel Pin [5] のフレームワークを用いて実装されている．解析対象プログラムが RDTSC 命令を呼ぶと，トレーサは CPU サイクル数の代わりにトレーサ内部で保持している整数値を返す．この整数値は RDTSC 命令が呼ばれるたびに 100 ずつ増加する仕組みになっている．したがって，CPU サイクル数の増加の仕方が実機と仮想マシンで異なることを利用したタイミング攻撃に対しては有効な手法であると言える．これに対し本研究は，RDTSC 命令に対して独自の整数値を返すという点においては類似しているが，実行した CPU 命令に要する CPU サイクル数に応じて整数値の増加量を動的に変化させている点で異なる．

FastVMIX [9] はバイナリ解析のためのシステムである．このシステムはタイミング攻撃の回避を主目的としたものではないが，RDTSC 命令を使用したタイミング攻撃を無効化する機能を有している．FastVMIX は BitVisor [22] というハイパーバイザをベースに開発され，ホストマシンと仮想マシンの間のコンテキストスイッチを排除している．解析対象プログラムが RDTSC 命令を呼ぶと，ハイパーバイザは解析にかかった時間だけ CPU サイクル数を減少させる．これによって，解析対象プログラムにとっては実機上で実行されたときと同じ時間経過になる．これに対し本研究は，タイミング攻撃の回避を主目的としており，より広範な攻撃手法に対して対策している点で異なる．

7. まとめと今後の課題

本研究では，マルウェアによるサンドボックス回避技術のうち，CPU サイクル数やスリープを使用するタイミング攻撃を回避する方法を提案した．提案手法を DynamoRIO のプラグインとして実装し，さらに解析回避処理の PoC プログラムを基にした仮想的なマルウェアを作成して実験したところ，ほとんどのタイミング攻撃を回避しつつ実行時間を大幅に短縮することに成功した．

今後の課題を述べる．まず，様々な時間の取得方法に対応することが重要である．本研究が対象とした手法は CPU サイクル数を用いた攻撃手法であり，これはシステム内部の時間である．したがって，システム外部の時間を取得して経過時間を計測する攻撃手法への対策をする必要がある．システム外部の時間の例として，NTP や HTTP の応答メッセージから取得できる時間が挙げられる．次に，実際のマルウェアに対する有効性を示すことである．本研究ではタイミング攻撃の要素技術のみを実装したものを仮想的なマルウェアとして実装し，これに対する有効性を示すことができた．しかし，Windows API の置き換えによる副作用によって，実際のマルウェアに対しては想定していなかった挙動が起きる可能性がある．最後に，マルチプラットフォームへの対応である．本研究は Windows 上でのタイミング攻撃に対してのみ有効性を示した．しかし，提案手法の基本的なアルゴリズムはプラットフォームに依存していない．近年，macOS や Linux を対象としたマルウェアが増加している [18] こともあり，様々なプラッ

トフォームに対して策を講じることは重要である．そのため，Windows 以外のプラットフォームでも動作するシステムの実装を行い，評価をすることが重要となる．

謝辞 本研究の一部は JSPS 科研費 20K11741 の助成を受けている．

参考文献

[1] 4. Instruction tables. https://www.agner.org/optimize/instruction_tables.pdf. 2021.08.09.

[2] Al-Khaser. https://github.com/LordNoteworthy/al-khaser. 2020.11.28.

[3] Cuckoo Sandbox. http://www.cuckoosandbox.org. 2020.01.20.

[4] DynamoRIO. https://github.com/DynamoRIO/dynamorio. 2020.11.28.

[5] Intel Pin. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html. 2020.11.28.

[6] McAfee Labs Threats Report. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-jun-2017.pdf. 2020.11.22.

[7] Paranoid Fish. https://github.com/a0rtega/pafish. 2020.11.28.

[8] tiny_tracer. https://github.com/hasherezade/tiny_tracer. 2021.08.07.

[9] 高速な VMI 機構を実装したバイナリ解析基盤. https://jinzaiipedia.ipa.go.jp/mitou_ipedia/wp-content/uploads/2020/05/3908349392e0324bdca7245b558ce362.pdf. 2021.08.07.

[10] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste. Malware Dynamic Analysis Evasion Techniques: A Survey. *ACM Computing Surveys (CSUR)*, 52(6):1–28, 2019.

[11] M. Brengel, M. Backes, and C. Rossow. Detecting Hardware-Assisted Virtualization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 207–227. Springer, 2016.

[12] A. Bulazel and B. Yener. A Survey on Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, pages 1–21, 2017.

[13] A. Chailytko and S. Skuratovich. Defeating Sandbox Evasion: How to Increase the Successful Emulation Rate in Your Virtual Environment. In *the 26st Virus Bulletin International Conference*, 2016.

[14] P. Chen, C. Huygens, L. Desmet, and W. Joosen. Advanced or Not? A Comparative Study of the Use of Anti-debugging and Anti-VM Techniques in Generic and Targeted Malware. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 323–336. Springer, 2016.

[15] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A Survey on Automated Dynamic Malware-Analysis Techniques and Tools. *ACM computing surveys (CSUR)*, 44(2):1–42, 2008.

[16] C. Kolbitsch, E. Kirda, and C. Kruegel. The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 285–296, 2011.

[17] L. Labs. Not So Fast My Friend – Using Inverted Timing Attacks to Bypass Dynamic Analysis. https://www.malwarebytes.com/resources/files/2020/02/2020_state-of-malware-report.pdf, 2014. 2021.08.12.

[18] Malwarebytes. 2020 State of Malware Report. https://www.malwarebytes.com/resources/files/2020/02/2020_state-of-malware-report.pdf. 2021.08.12.

[19] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach. Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. *ACM Computing Surveys (CSUR)*, 52(5):1–48, 2019.

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

[21] H. Shi, J. Mirkovic, and A. Alwabel. Handling Anti-Virtual Machine Techniques in Malicious Software. *ACM Transactions on Privacy and Security (TOPS)*, 21(1):1–31, 2017.

[22] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, et al. Bitvisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, 2009.

[23] 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**, 2017(2), 2017.