Computer Security Symposium 2017
23 - 25 October 2017

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

大山 恵弘[1]

概要：多くのマルウェアが様々な目的でスリープを実行する．動的マルウェア解析では，解析にかかる時間の短縮のために，スリープの実行がしばしばスキップされる．ただし，マルチスレッドのマルウェアの解析では，スリープのスキップは並行実行の非決定性に起因する挙動の変化をもたらす可能性があるため，適用して良いかどうかは簡単には判断できない．本研究では，マルチスレッドのマルウェアの挙動に与える影響をできるだけ小さくしながら，スリープのスキップやスリープ時間の短縮を実現する方式を示す．

キーワード：マルウェア，動的解析，耐解析処理，スリープ，マルチスレッド，競合条件

A Method of Shortening Sleep Duration in Dynamic Malware Analysis

Yoshihiro Oyama[1]

Abstract: Many malware programs execute sleeps for various purposes. In dynamic malware analysis, executions of sleeps are often skipped to shorten the analysis time. However, in the analysis of a multithread malware program, determining whether a given sleep can be skipped is not straightforward because the skip can change the behavior of the program owing to the nondeterminism of concurrent execution. In this study, we present a method of skipping sleeps or shortening sleep duration in multithread malware programs with the effect on the behavior minimized.

Keywords: Malware, dynamic analysis, anti-analysis operations, sleeps, multithread, race condition

1. はじめに

現代的なマルウェアの多くは解析を妨害するための処理（**耐解析処理**）を実行する [3,14]．例えば，一部のマルウェアは，自身がマルウェアを解析するためのシステム（**解析システム**）上で動作しているかどうかを検査し，そうであった場合には実行を終了させたり変化させたりする．また，長時間のスリープを実行することにより，動的解析をタイムアウトで中断させるマルウェアも存在する [18]．今日，高い精度や速度で動的解析を行うには，解析システムに耐解析処理への対策を組み込むことは必須である．

著者は，高い優先度での対策が必要な耐解析処理は 2 つあると考えている．1 つは解析システムを検出する処理であり，もう 1 つはスリープである．動的解析において，解析システムの存在をマルウェアに検出させない，または，検出させたとしても動作を変えさせないことは，解析結果を妥当にする点で重要である．また，動的解析に要する時間をスリープによって長大化させないことは，解析を実用的かつ効率的にする点で重要である．

多くの解析システムには，スリープの実行を飛ばす（スリープをスキップする）機構が組み込まれている．この機構により，マルウェアが長時間のスリープを実行しても，解析時間を短く保つことができる．例えば，広く利用されている解析システムである Cuckoo Sandbox [4]（以下，Cuckoo）にも，この機構が組み込まれている．しかし，この機構は常に利用できるわけではない．マルチスレッドプログラムや時間情報を取得するプログラムでは，スリープのスキップはプログラムの挙動ひいては動的解析の結果を変えうる．Cuckoo も，プログラムがスレッドを生成したらスリープを全くスキップしなくなる．プログラムの挙動をできるだけ変えずに，スリープのスキップやスリープ時間

[1] 筑波大学
University of Tsukuba

© 2017 Information Processing Society of Japan

の短縮を実現することができれば，高い効率で妥当な解析結果が得られる．しかし，著者の知る限り，マルチスレッドプログラムが実行するスリープに対してスキップまたは時間の短縮をするための方式を示した研究は存在しない．

本論文では，マルチスレッドで実現されたマルウェアを，挙動をできるだけ変えないようにしながら，スリープの時間を短縮して実行する方式を提案する．その短縮は，スリープ時間を 0 にする（スリープをスキップする）ことも含む．さらに，Linux プログラムを対象に，その方式に基いてスリープをスキップするシステム FarFuture を実装して実験を行った結果を報告する．FarFuture の実装には，Intel が開発した dynamic instrumentation tool である Pin [13] を用いた．実験では，時間情報や実行のタイミングを利用して解析システムの存在を検出するプログラムを用いた．FarFuture により，それらのプログラムでスリープ時間の短縮を，検出されないようにしながら実現できることを確認した．

2. 耐解析処理

本研究が対象とする耐解析処理の概要を説明する．

2.1 解析システムの検出

一部のマルウェアは，マルウェア解析に用いられるサンドボックス，デバッガ，ハイパバイザなどのシステムを検出する処理を実行する．検出に用いられる手段 [3, 15, 16] のうち主なものを以下に示す．本研究ではそれらのうち時間情報の検査による耐解析処理のみを扱う．

資源の名前や属性の検査　資源の名前や属性，例えばハードディスクの名前やネットワークインタフェースの MAC アドレスが，特定のハイパバイザやサンドボックスを示唆することがある．また，所定の名前のファイルやレジストリの有無や内容，同時に走っているプロセスやサービスが，特定の解析システムを示唆することがある．

プロセスの状態やコードの検査　デバッグレジスタの値，プロセスのトレース状態，リンクされたライブラリ，メモリ上にロードされたコード（の書き換えの有無）などの検査により，サンドボックスやデバッガの存在を推定できることがある．

時間情報の検査　解析システム上では元の環境と比べて特定の処理に要する時間が大きく変化することがあり，この変化の検出によって解析システムを検出できる．例えば，CPUID 命令の実行に要するサイクル数が大きく変化することを利用してハイパバイザを検出する方法が広く知られている．また，短い処理の実行に長い時間がかかることがデバッガのブレークポイントの存在を示唆したり，逆に長い処理の実行が短い時間で終わることが，サンドボックスによる制御や時間情報の修正を示唆したりすることがある．さらに，プロセスに割り当てられる CPU 時間やコンテキストスイッチの挙動を利用しても，解析システムの存在を推定できることがある [20]．

2.2 解析時間の長大化

マルウェアは長時間のスリープによって動的解析に要する時間を伸ばすことができる．動的解析では多くの場合にタイムアウトを設けている．例えば Cuckoo ではデフォルトではタイムアウトは 120 秒である．マルウェアはタイムアウトよりも長い時間のスリープを実行することにより，自身の挙動についての情報をほとんど与えないまま動的解析を中断させることができる．

2.3 スリープのスキップを悪用した解析システムの検出

マルウェアはスリープのスキップを悪用して，解析システムを容易かつ効果的に検出することができる．スリープをスキップすると，スリープ後の処理が実行されるタイミングが変わるが，それがプログラムのタイミング以外の挙動まで変化させることがある．その変化を検出すれば，解析システムを検出することができる．変化する挙動のうち自明なものには，時間情報を取得するプログラムの挙動がある．また，マルチスレッドプログラムでは，時間情報を取得していなくても，実行の非決定性によって挙動が変化することがある．

図 1 に，スリープのスキップによって出力が変わるプログラム（言語は C 言語で OS は Linux，以降も同様）を示す．スリープによってスレッド間の競合や実行の非決定性が事実上排除されているため，通常の実行では，このプログラムはほぼ確実に 1 を出力する．ここで，このプログラムが解析システム上で実行され，タイムアウトを招く `sleep(3600)` がスキップされる（かつ，`sleep(1)` はスキップされない）場合を考える．その場合，このプログラムはほぼ確実に 2 を出力する．解析システム上での実行結果が実環境での実行結果と異なることは，解析結果の妥当性を下げる面でも，解析システムを検出する手がかりをマルウェアに与える面でも，望ましくない．しかし，`sleep(3600)` を忠実に実行すると，`thread1` の実行は 3600 秒も停止し，その後の処理が解析できるまで長時間待たなければならない．

上記の問題は，一般化すれば，並行プログラムにおける競合を用いた，時間情報の非明示的な取得と利用である．スリープのスキップが変化させるのは時間情報であるが，プログラムは時間情報を明示的に取得しなくても処理のタイミングから時間情報を「取得」できることが，この問題を招いている．実行時間情報を「取得」できるプログラムにとっては，スリープのスキップやスリープ時間の修正はプログラムの挙動を変化させるものとなる．

© 2017 Information Processing Society of Japan

```
volatile int a = 1;

void *thread1(...)
{
  sleep(3600);
  a = 2;
  ...
}

void *thread2(...)
{
  sleep(1);
  printf("%d\n", a);
  ...
}

int main(...)
{
  ...
  pthread_create(..., thread1, ...);
  pthread_create(..., thread2, ...);
  ...
}
```

図 1 スリープのスキップによって出力が変わるプログラム

3. 既存システムでの方式

プログラムの挙動を変えずにスリープ時間をできるだけ短いものに修正して実行することは容易ではない．一般に，スレッドは他のスレッドがどのタイミングで自身と相互作用するかを，その事象の発生前に常に正確に知ることはできない．その結果，スリープからの復帰をどの時間範囲内に実行すれば元の挙動が維持されるかも知ることができない．

動的解析のためのシステムの多くは商用製品であり，それらがスリープにどう対処しているかについての詳細は明らかにされていない．一方，Cuckoo はオープンソースである．Cuckoo は，内部処理が明らかにされており，かつ，スリープのスキップが実装されている数少ないシステムの 1 つである．そこで以下では，Cuckoo がスリープをスキップする方式を説明する．

Cuckoo は以下の両条件を満たすときにのみスリープをスキップする [1].

(1) マルウェアの実行開始から 5 秒以内である

(2) マルウェアがプロセスもスレッドも生成していない

条件 (1) は本研究で扱う問題とは関係がない．条件 (2) は，スリープのスキップがマルウェアの挙動を変化させる可能性を考慮して導入されていると考えられる．この条件により，Cuckoo は，複数のスレッドやプロセスからなるプログラムが実行するスリープをスキップしない．例えば，図 1 のプログラムの `sleep(3600)` をスキップしない．その結果，Cuckoo は，マルウェアが試みる長時間のスリープを忠実に実行することが多く，タイムアウトで動的解析が中断することも多いという問題を持っている．

4. 提案方式

4.1 前提

本研究では競合として，メモリを通じたスレッド間の競合のみを考える．他にも，ファイルなどの資源を通じたプロセス間の競合などの多くの競合があるが，それらはシステムコールを伴うことが多く，解析システムがより把握や制御をしやすいと考えるため，本研究では扱わない．

プログラムの挙動が維持されるかどうかを適切かつ厳密に定義することは必ずしも容易ではない．本研究では，プログラムの出力が同一であれば挙動が維持されているとみなすこととする．

現在は，マルウェアも提案方式も Linux 上で動くことを仮定している．提案方式を若干修正すれば，Windows 上での実行に対応することも可能であると推測している．

4.2 基本アイデア

スリープによって実行の非決定性が事実上排除されているプログラムでは，挙動を高い可能性で維持したまま，スリープのスキップやスリープ時間の修正が実現できることがある．図 1 のプログラムはその一例であるが，より複雑な例を**図 2** に示す．このプログラムを実環境で実行すると，スリープが実行の非決定性に与える効果により，ほぼ必ず 1, 2, 3, 4, 5 が順に表示される．一方，全てのスリープをスキップすると，スレッド間の競合により，1, 1, 3, 4, 4 や 1, 3, 3, 3, 6 などの様々な順で数が表示される可能性がある．

各スレッドの実行状態に基いて，適切な順番でスリープ時間を適切な値に修正する機能を有している解析システムであれば，このプログラムの元の挙動を維持しながら，図中の全てのスリープをスキップすることができる．このプログラムではまず，`thread1` が 10 秒のスリープを実行し，`thread2` が 20 秒のスリープを実行する．この時点で，最初のスレッドを含む 3 スレッド全てが，スリープやスレッドの終了を待つ停止状態に入る．このように全スレッドが停止状態にある時には，元の挙動を高い可能性で維持しながら，スリープをスキップすることができる．具体的には，最初にどれかのスレッドが停止状態から抜ける予定時刻まで，全スレッドの「時間を進める」こと，すなわち，全スレッドの残りスリープ時間などをその時刻までの時間だけ減らすことができる．上記の状態では，10 秒を進めることができる．すなわち，`thread1` のスリープを即座に終了させ，`thread2` の残りスリープ時間を 20 秒から 10 秒に短縮することができる．その後，実行を再開した `thread1` はすぐにまたスリープを実行して，再び全スレッドが停止状態に入るが，今度は `thread2` のスリープを即座に終了させることができる．以降も同様の処理が繰り返される．

© 2017 Information Processing Society of Japan

```
volatile int a = 1;

void *thread1(...)
{
  sleep(10);

  printf("%d\n", a);
  a = 3;
  sleep(20);
  printf("%d\n", a);
  a = 5;
  sleep(20);
  ...
}

void *thread2(...)
{
  printf("%d\n", a);
  a = 2;
  sleep(20);
  printf("%d\n", a);
  a = 4;
  sleep(20);
  printf("%d\n", a);
  a = 6;
  ...
}

int main(...)
{
  ...
  pthread_create(&th1,
           thread1, ...);
  pthread_create(&th2,
           thread2, ...);
  pthread_join(th1, ...);
  pthread_join(th2, ...);
  ...
}
```

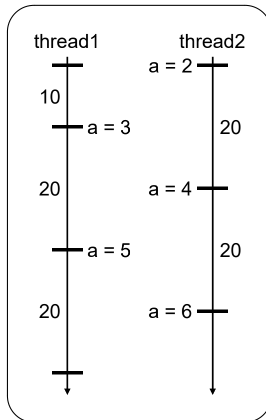

図 2 実行の非決定性が事実上排除されているプログラムと，各スレッドによる処理のタイミング

全スレッドが停止状態にあることは，プログラムの挙動を高い可能性で維持しながらスリープ時間を短縮できることの 1 つの十分条件である．提案方式ではこの条件を利用する．挙動を維持するには，元の実行で 2 つの処理の間に実行順序関係があるときに，それを変えないことが重要である．停止している時間を全スレッドから同時に同じ幅だけ減らす操作は，以後の各処理の相対的な実行タイミング，ひいては実行順序関係を変えない可能性が極めて高い．

マルウェアがシングルスレッドで動いている場合には，スリープを実行すると必ず「全スレッド」が停止状態に入るので，そのスリープは必ずスキップされる．すなわち，Cuckoo の条件 (2) は，この条件が満たされるケースの部分ケースに入っているかどうかを判定する別の十分条件であるとみなせる．

4.3 設計

提案方式では，マルウェアなどのプログラムを別のプログラムで監視しながら実行する．監視されるプログラムを**対象プログラム**，監視するプログラムを**モニタ**と呼ぶ．

モニタは，対象プログラムの全スレッドが停止状態に入ったら，実行中の全てのスリープに対してスリープ時間の修正を適用する．まず，最初に終了する予定のスリープの残りスリープ時間を得る．その時間を**スキップ時間**と呼ぶ．次に，実行中の全てのスリープの残りスリープ時間から，スキップ時間を引く．よって，最初に終了する予定のスリープは即座に終了する（スキップされる）．スレッドの終了やロック解放などのイベントを待っているスレッドに対しては何もしない．

モニタは，インターバルタイマの時間も修正する．スリープ時間と同様に，インターバルタイマの満了までの時間からもスキップ時間が引かれる．

モニタは対象プログラムにおけるスレッドの開始と終了を捕捉し，実行中のスレッドの情報を管理する．加えて，対象プログラムが実行する以下のシステムコールも捕捉し，必要に応じてそれらの動作を変える．

- スリープのためのシステムコール：Linux では `nanosleep` などである．
- 時間情報を取得するシステムコール：Linux では `clock_gettime` などである．
- 同期のためのシステムコール：Linux では `futex` などである．
- インターバルタイマを操作するシステムコール：Linux では `setitimer` などである．

アプリケーションプログラムはシステムコールよりも高い抽象度の関数（`sleep` や `pthread_join` など）を呼び出すことが多いが，通常，それらの関数の中では結局これらのシステムコールが呼び出される．例えば，本研究で用いた実験環境では，ロック，セマフォ，条件変数を操作する関数を実行すると，`futex` システムコールが呼び出される．

上記に挙げた以外にも，タイムアウトを指定できるシステムコール（例えば `select` や `rt_sigtimedwait`）も，事実上のスリープの実行に利用できるので，捕捉して動作を変えることが望ましい．しかし現在は，そのようなシステムコールにも対処するには大きな手間がかかるため，少数の重要なシステムコールのみに対処するようにしている．

モニタは対象プログラムが取得する時刻情報も修正する．モニタは，それまでに使用したスキップ時間の和（**累積スキップ時間**）を保持し，取得された時刻を，累積スキップ時間を加えた時刻に修正する．また，`nanosleep` システムコールなどのいくつかのシステムコールでは，スリープがシグナルに割り込まれると残りスリープ時間が呼び出し元に返されるが，それも適切な値に修正する．

4.4 実装

4.4.1 Pin

提案方式を実現するシステム FarFuture を，Pin [13] を用いて実装した．Pin はプログラムのバイナリコードを動的に書き換えることによって，プログラムに処理を追加す

© 2017 Information Processing Society of Japan

る（instrument する）ことを可能にする．FarFuture は，スレッドの開始と終了の部分およびシステムコールの入口と出口に処理を追加した上で，対象プログラムを実行する．

Pin はバイナリコードを instrument するため，対象プログラムのソースコードは必要がない． また，Pin は Windows プログラムも Linux プログラムも instrument することができる．これらの 2 点はマルウェア解析において極めて重要であり，本研究で Pin を用いた理由となっている．

FarFuture は Pin 用の C++言語のコードで実装されている．そのコードは共有ライブラリのファイル（.so ファイル）にコンパイルされ，例えば以下のようなコマンドで実行される．

```
$ pin -t farfuture.so -- ./malware
```

マルウェアの実行型ファイル `malware` が，共有ライブラリ `farfuture.so` のコードで instrument された上で実行される．

提案方式を実現するコードを解析システムに組み込むのではなく，Pin 用の独立した共有ライブラリとして提供する理由は，様々な解析システムと組み合わせて使えるという利点を重視するためである．ただし，FarFuture のコードは対象プログラムと同一のプロセスアドレス空間内で動作するので，例えば仮想マシンモニタ内に実装されたコードに比べて，その存在をマルウェアに検出されやすかったり，処理を無効化されやすいという欠点もある．

4.4.2 モニタ

モニタはシステムコールを捕捉して情報を取得し，以下のデータ構造を維持する．

スレッドテーブル　全スレッドについて，実行状態および，スリープを実行中ならばそのスリープの情報が格納されている．実行状態は，実行中，スリープ中，イベントの待機中のどれかである．スリープの情報は，スリープ開始時刻と，当初指定された（修正されていない）スリープ時間からなる．

起床時刻テーブル　現在実行中であるスリープおよび現在有効であるインターバルタイマの情報が格納されている．スリープについては，実行したスレッドの ID とスリープの終了時刻が格納されている．インターバルタイマについては，タイマが満了する時刻とタイマの周期が格納されている．

Linux では一旦実行が開始したスリープの残りスリープ時間は，ユーザレベルでは，シグナルでスリープを中断させて 0 にする以外の手段で修正することはできない．よって，FarFuture には，残りスリープ時間を任意の時間に修正するための処理が実装されている．対象プログラムがスリープのためのシステムコールを呼び出したら，モニタはシステムコール番号の付け替えにより，そのシステムコールを `futex` システムコールに変える．元のシステムコールに与えられていた引数の情報はスレッドテーブルと起床時刻テーブルに記録する．スリープの開始と終了は，同期用データ構造を介しての通知の待機と受信に変換される．モニタは `futex` システムコールの引数に自身が作成した同期用データ構造を与え，`futex` システムコールは，そのデータ構造に通知が来るまでタイムアウトなしで待つ．モニタは適切な時刻にそのデータ構造に通知を送る．

一方，インターバルタイマの満了までの時間の修正については，Linux では新しいタイマの値を設定すればそれが有効になるため，特別な処理は実装していない．モニタが自らタイマの値を設定するシステムコールを呼び出して修正している．

起床時刻テーブルの情報が更新された際には，モニタは，全スレッドがスリープ中またはイベントの待機中の状態（停止状態）であるかどうかを検査する．その検査の結果に応じて以下の片方を実行する．

- 全スレッドが停止状態にある場合には，起床時刻テーブルに格納されている時刻のうち最も早い時刻までの時間をスキップ時間とし，スリープ時間を短縮する．具体的には，起床時刻テーブルに格納されたスリープの終了時刻とタイマが満了する時刻を全て，スキップ時間分だけ早い値に修正する．次に，その修正によって実行が再開されるスレッドが待つ同期用データ構造に通知を送る．
- 停止状態にないスレッドがある場合には，モニタは，起床時刻テーブルに格納されている時刻のうち最も早い時刻までの時間を，インターバルタイマの満了までの時間として設定する．満了したら OS は対象プログラムではなくモニタにアラームシグナルを送信するようにしておく．モニタは，自身が準備したアラームシグナルのハンドラにおいて，終了したスリープまたは満了したインターバルタイマの情報を起床時刻テーブルから取り除く．それがスリープである場合には，その終了を待つための同期用データ構造に通知を送る．それがインターバルタイマである場合には，次の周期についての情報を起床時刻テーブルに追加するとともに，モニタから対象プログラムにアラームシグナルを注入する．その結果，対象プログラムが準備したアラームハンドラが実行される．

現状ではマルチプロセスへの対処は未実装である．対象プログラムが `fork` システムコールなどによるプロセス生成を試みたら，対象プログラムを即座に終了させている．一方，`execve` システムコールなどによる新規プログラムの実行への対処はすでに実装されている．新規プログラムの実行の際には，スレッドテーブルと起床時刻テーブルが初期化される．

4.5 議論

スリープ時間を修正する条件としては様々なものが考え

© 2017 Information Processing Society of Japan

```
volatile int a;

void *thread1(...)
{
  sleep(100);
  a = 1;
  ...
}

void *thread2(...)
{
  long_computation();
  a = 2;
  ...
}

int main(...)
{
  ...
  pthread_create(&th1, thread1, ...);
  pthread_create(&th2, thread2, ...);
  pthread_join(th1, ...);
  pthread_join(th2, ...);
  printf("%d\n", a);
  ...
}
```

図 3　提案方式でスキップできないスリープ

られる．Cuckoo が採用している「シングルスレッドシングルプロセスであること」は，かなり厳しい条件であり，多くのスリープをスキップできなくなる．より緩い条件としては「スレッド間やプロセス間に相互作用が無いこと」が考えられる．異なるスレッドやプロセスが共通のメモリ領域や計算資源にアクセスしないことや書き込みをしないことを，静的解析などの方法で保証できる場合には，相互作用が無いと判断できる可能性がある．しかし，その保証は必ずしも容易ではない．

提案方式では「全スレッドが停止状態にあること」を条件として採用している．この条件には，マルチスレッドのマルウェアによるスリープもスキップできることや，上記の相互作用についての保証を必要としないという利点がある．一方，少なくとも 1 スレッドが計算を実行している限り，スリープの時間を全く短縮できないという欠点がある．そのようなことが起きるプログラムの例を**図 3** に示す．このプログラムでは，`sleep(100)` と `long_computation()` のどちらが早く終わるかによって実行結果が変わる．`long_computation()` の計算にかかる時間が 100 秒より長いか短いかは実行しない限りわからないものとする．この場合，この 100 秒のスリープ時間を別の値に修正することは困難である．もし，`a = 1` と `a = 2` の 2 行が無く，その結果 `thread1` と `thread2` の間の相互作用も無いならば，`sleep(100)` をスキップしてもプログラムの挙動は変化しないかもしれない．しかし，現状の提案方式では相互作用の有無を考慮していない．

提案方式ではシステムコールによって取得された時間情報のみを修正しているが，これについては今後検討が必要である．時間情報は他の多くの手段でも取得できる．例えば，RDTSC 命令などの時間に関係する機械語命令の実行や，ネットワークを介した時刻の問い合わせにより，マルウェアが正確な時間を把握する可能性がある．それらの方法への対処も今後考える必要がある．

5.　実験

5.1　FarFuture を用いた実験

FarFuture の動作を確認する実験を行った．実験の環境は CentOS 6.9 (kernel 2.6.32-504.12.2.el6.x86_64), Intel Xeon E5-2650, 32GB RAM である．Pin のバージョンは 3.2，コンパイラは gcc 4.4.7 である．

マルウェアによる解析システムの検出処理を模した処理を実行するプログラム P1, P2, P3 を，FarFuture で監視しながら実行した．それらのプログラムの概要と実行結果を以下に示す．

耐解析プログラム P1：　1 秒のスリープ，2 秒のスリープ，...，10 秒のスリープを順に実行する．各スリープの実行の前後で `clock_gettime` システムコールによって時刻を取得する．各スリープの実行後に，スリープ時間と実際に経過した時間を比較し，大きい方が小さい方の 1.1 倍以上である場合には解析システムが存在するとみなす．

P1 の実行結果：　FarFuture は全てのスリープをスキップできた．かつ，実行結果は元のものから変化しなかった（どちらの場合にも解析システムを検出しなかった）．また，スリープ時間を短縮するが，対象プログラムが取得する時間情報を修正しない設定で FarFuture を用いた場合には，対象プログラムは解析システムを検出した．プログラムの実行時間は元々約 55 秒だったが，FarFuture を用いた場合には 1 秒未満だった．

耐解析プログラム P2：　2 スレッド（スレッド A，スレッド B）を生成する．両スレッドとも，整数型の共有変数 S の値を 1 増やし，直後に 200 ミリ秒のスリープを実行するという処理を 100 回ずつ実行する．なお，スレッド B だけはスレッド開始直後に 100 ミリ秒のスリープを実行する．この 100 ミリ秒のスリープにより，両スレッドは S の値を交互に更新することになり，合計 200 回の更新の実行順は事実上決定的になる．S の初期値は 0 である．スレッド A は S の値が偶数ならば 1 増やし，奇数ならば解析システムが存在すると判断して即座に実行を終了する．スレッド B は奇偶に関して反対の処理を実行する．よって，更新が 1 回でも交互にならない場合には，解析システムが検出される．

P2 の実行結果：　FarFuture は全てのスリープのスリープ時間をほぼ 0 秒に短縮できた．かつ，実行結果は元のものから変化しなかった（どちらの場合にも解析システムを検出しなかった）．具体的には，FarFuture は全ての 200 ミリ秒のスリープにおいて，まず約 100 ミ

© 2017 Information Processing Society of Japan

リ秒分を短縮して残り時間を約 100 ミリ秒とした．同様に，残り時間が約 100 ミリ秒となった全てのスリープにおいて，その時間全体を短縮して残り時間を 0 秒とした．両スレッドに対してこれらの処理を繰り返し実行した．プログラムの実行時間は元々約 20.1 秒だったが，FarFuture を用いた場合には 1 秒未満だった．

耐解析プログラム P3： インターバルタイマを用いて，4 秒後にアラームシグナルが送信されるように設定する．アラームシグナルのハンドラでは T に 1 が代入される．設定した直後に 10 秒スリープし，変数 T に 2 を代入する．さらに，十分長い計算を実行した後に T の値を読む．T の値が 1 であれば，スリープ後の代入がハンドラよりも先に実行された可能性があり，解析システムが存在するとみなす．

P3 の実行結果： FarFuture はインターバルタイマの満了までの時間とスリープの時間をほぼ 0 秒に短縮することができた．かつ，実行結果は元のものから変化しなかった（どちらの場合にも解析システムを検出しなかった）．具体的には FarFuture はまずタイマの 4 秒分の時間を進めた．すなわち，タイマの満了までの時間を約 0 秒に短縮し，スリープの残り時間を約 6 秒に短縮した．その短縮処理の直後にアラームシグナルが送信され，ハンドラが実行された．ハンドラではスリープの残り時間が 0 秒に短縮された．また，スレッドの実行状態を考慮せず全てのスリープを常にスキップしつつ，インターバルタイマは忠実に実行する設定で FarFuture を用いた場合には，T の値は 1 となり，対象プログラムは解析システムを検出した．

まとめると，これらの実験により，FarFuture は時間情報を用いて解析システムを検出する処理に対して

- スリープの終了やインターバルタイマの満了までの時間を短縮して高速に実行し，かつ，
- プログラムの元の挙動を維持した

ことを確認した．

さらに，FarFuture が対象プログラムの実行時間に加えるオーバヘッドを計測した．対象プログラムとして，1 MB のファイルをコピーする処理を 1000 回実行するプログラムを用いた．このプログラムはファイル入出力のシステムコールを非常に多く実行するが，時間やスリープに関係するシステムコールを 1 度も呼び出さず，スレッドもプロセスも生成しない．元の実行時間は約 10.0 秒であり，FarFuture を用いた場合の実行時間は 11.7 秒であった．FarFuture はある程度のオーバヘッドをもたらすが，多くのシステムコールを呼び出すプログラムであっても，解析が非現実的になるほどには実行時間は伸びない．

5.2 既存の解析システムを用いた実験

前述の P1, P2, P3 とほぼ同じ処理を実行する Windows プログラム Q1, Q2, Q3 を Cuckoo の上で実行した．使用したゲスト OS は Windows XP (32 bit) である．ビルドには Visual Studio 2015 を用いた．スリープの実現には `Sleep` 関数を用いた．

Q1 では `timeGetTime` 関数を用いて時刻を取得した．Q1 では，スリープのスキップの結果，`Sleep` 関数に与えられたスリープ時間と実際に経過した時間が大きく食い違い，解析システムが検出された．

Q2 では，スレッドが生成されたために，スリープは 1 回もスキップされなかった．その結果，実行結果は元のものから変化せず，Q2 は解析システムを検出しなかったが，解析には元の実行時間とほぼ同じ時間がかかった．

Q3 としてはタイマの実現に `CreateTimerQueueTimer` 関数を使う版 Q3A と `SetTimer` 関数を使う版 Q3B を実装した．Q3A ではタイマの処理のために暗黙にスレッドが生成されたため，検出と解析時間に関する結果は Q2 と同じであった．Q3B は，スレッドを生成させず，タイマからのメッセージをポーリングしながら小刻みの時間のスリープを繰り返し実行する形で，プログラムを記述した．Q3B ではスリープのスキップの結果，解析システムが検出された．

6. 関連研究

マルウェアが実行する長時間のスリープが大きな問題であることはマルウェア解析の分野で広く知られている．長時間のスリープの実行を検出してユーザに伝えたり，そのようなスリープをスキップしたりするサンドボックスも多い．マルチスレッドプログラムにおいてスリープをスキップすると挙動が変わりうる問題については既に複数の指摘がある [1,2,9,11]．しかし，この問題に対する対策を提案した研究は著者の知る限り存在しない．本研究では，単純ではあるが一定の効果がある対策を提案している．

マルチスレッドプログラムにおいてタイミングによって挙動が変化する処理を検出する技術は競合検出と呼ばれ，それについては多くの研究がある．古くは Lamport らの研究 [10] でこの技術が扱われた．その後も Eraser [17]，DataCollider [5]，Portend [7] などの研究が存在する．競合検出を行う実用的なソフトウェアも多く存在し，例えば，Valgrind [19] に基づくツールである Helgrind や，Intel Inspector [6] がある．競合検出に関する既存技術の多くは，挙動が変化する可能性がある処理を検出することはできるが，そのような処理をどういうタイミングで実行すれば挙動を維持できるかという問題に対する答を与えていない．

Dthreads [12] は，マルチスレッドプログラムを決定的に実行するためのスレッドライブラリである．Dthreads を用いると，競合があるプログラムであっても，同じ入力に対しては毎回同じ出力が出されるようになる．動作を維持しながらスリープ時間を修正する用途に対して Dthreads のような仕組みは有効である可能性があるが，実際に有効

© 2017 Information Processing Society of Japan

であるかは明らかにされていない．今後，本研究の文脈におけるこのような仕組みの有効性を検証する必要がある．

Hasten [8] は実行の進行を遅らせる耐解析処理による影響を軽減する動的解析システムである．Hasten は API 呼び出しなどの一定の処理を繰り返して時間を経過させる処理を検出し，その処理からの早期の脱出を試みる．本研究と異なり，Hasten はスリープを対象としていない．

長時間のスリープを含む各種の耐解析処理をどの程度の割合のマルウェアが実行するかを明らかにした研究 [14,21,22] が存在する．また，マルウェアが実行するスリープ挙動の傾向を示し，スリープ挙動を手がかりにマルウェアの分類や検出を試みる研究 [21] も存在する．これらの研究は動的解析結果を利用する技術を提案するものであり，動的解析を効率的にする技術を提案するものではない．

7. まとめと今後の課題

本研究では，マルチスレッドのマルウェアの挙動をできるだけ変えないようにしながら，スリープ時間やインターバルタイマの満了までの時間を短縮する方式を提案した．その方式を実現するシステムを Linux 上に実装し，時間情報や実行のタイミングを利用した耐解析処理を実行するプログラムに対して，そのシステムが意図通りに動作することを確認した．さらに，同様の動作をするプログラムを既存の解析システム上で実行すると，挙動が変化したり，スリープのスキップが抑制されたりすることを確認した．

今後の課題としては，第一に，競合検出の分野での研究成果を取り入れて提案方式を発展させることが挙げられる．例えば，競合条件の有無を高い精度で推定し，その推定に基づいてスリープ時間を修正する拡張が考えられる．第二に，FarFuture を Windows 上でも動作するようにし，Windows 向けのマルウェアも扱えるようにすることが挙げられる．第三に，RDTSC 命令による時間情報の取得などの，時間に関連する耐解析処理のうち本研究では扱わなかったものへの対処が挙げられる．これらの処理はシステムコールに比べて捕捉がより難しいとともに，これらの処理で取得される時間情報の補正もより複雑である可能性が高く，対処は挑戦的で興味深い課題であると考えている．

謝辞 本研究において，株式会社富士通研究所の小久保博崇氏より有用な助言をいただいた．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] Cuckoo mailing list, `http://public.honeynet.org/pipermail/cuckoo/2013-March/001354.html`.

[2] Balazs, Z.: Malware Analysis Sandbox Testing Methodology, *The Journal on Cybercrime & Digital Investigations*, Vol. 1, No. 1 (2016).

[3] 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).

[4] Cuckoo Sandbox, `https://cuckoosandbox.org/`.

[5] Erickson, J., Musuvathi, M., Burckhardt, S. and Olynyk, K.: Effective Data-Race Detection for the Kernel, *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (2010).

[6] Intel: Intel Inspector, `https://software.intel.com/en-us/intel-inspector-xe`.

[7] Kasikci, B., Zamfir, C. and Candea, G.: Data Races vs. Data Race Bugs: Telling the Difference with Portend, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 185–198 (2012).

[8] Kolbitsch, C., Kirda, E. and Kruegel, C.: The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code, *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pp. 285–296 (2011).

[9] Kruegel, C.: Evasive Malware Exposed and Deconstructed, RSA Conference 2015 (2015).

[10] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565 (1978).

[11] Lastline Labs: Not so fast my friend - Using Inverted Timing Attacks to Bypass Dynamic Analysis, `http://labs.lastline.com/not-so-fast-my-friend-using-inverted-timing-attacks-to-bypass-dynamic-analysis` (2014).

[12] Liu, T., Curtsinger, C. and Berger, E. D.: Dthreads: Efficient Deterministic Multithreading, *Proceedings of the 23rd ACM Symposium on Operating System Principles*, pp. 327–336 (2011).

[13] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. and Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 190–200 (2005).

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

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

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

[17] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. and Anderson, T.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs, *ACM Transactions on Computer Systems*, Vol. 15, No. 4, pp. 391–411 (1997).

[18] Singh, A. and Bu, Z.: Hot Knives Through Butter: Evading File-based Sandboxes, Technical report, FireEye (2014).

[19] Valgrind, `http://www.valgrind.org/`.

[20] 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).

[21] 大山恵弘：マルウェアのスリープ挙動の多様性に関する予備調査，情報処理学会研究報告コンピュータセキュリティ，Vol. 2017-CSEC-76 (2017).

[22] 大山恵弘：マルウェアによる対仮想化処理の傾向についての分析，コンピュータセキュリティシンポジウム 2016 論文集，pp. 534–541 (2016).

© 2017 Information Processing Society of Japan