仮想マシン内部情報を簡潔かつ対話的に取得可能なシステム

浅見 祥喜[1,a)]　岩崎 英哉[1,b)]

概要：オペレーティングシステム (OS) 内部からでは検出が困難なマルウェアを検出するために，仮想マシンモニタ (VMM) を用いて，別の仮想マシン (VM) や VMM 層から監視対象 OS のメモリなどを参照する手法がある．LibVMI は，実行中の VM のメモリの内容を参照する機能等を提供するライブラリであるが，これを利用して対象 OS を監視するには，LibVMI が提供するライブラリ関数を直接利用したプログラムを監視内容ごとに作成し，コンパイル，実行する必要がある．しかし，対象 OS 固有の構造体メンバのオフセット等のメモリ配置を強く意識した細かな計算処理などを書く必要がありプログラムが煩雑になるため，LibVMI を用いてマルウェアを検出を試みる場合，ユーザに大きな負担がかかる．この問題を解決するため，本研究では対話的に VM 内部情報を取得可能なシステムを提案する．VM 内部情報取得のための専用の言語を提供することで，上記のような煩雑なプログラミング作業は不要である．また，対話的に利用することができるため，即時性が高く必要に応じた VM 内部検査が可能である．

1. はじめに

近年，ウイルスやワームといった悪質なソフトウェア (マルウェア) による被害が増大しており，マルウェア対策として様々な手法が開発されている．

コンピュータ内部に侵入したマルウェアを検知するために，オペレーティングシステム (OS) 内部で動作する検知システムを導入する，という方法がある．しかし，マルウェアに感染したシステム内でのマルウェア検知は，検知システムやその他の情報が改竄されている可能性があることから，この方法には検知結果が必ずしも信用できないという大きな問題がある．例えば，ELF_XORDDOS.AR [1] は OS 内のシステムを変更することで，自身に関連したファイルやプロセスを隠蔽することで，検知システムによる検出を防ぐ機能を持つマルウェアであり，上述したようなマルウェア検知方法では検知できない．また，侵入したマルウェアによって検知システムの動作を無効化される可能性もある．これらの問題への対処方法のひとつとして，仮想マシンモニタ (VMM: Virtual Machine Monitor) など監視対象 OS の外部に検知システムをおく方法がある．

VMM とは，仮想マシン (VM) を制御・管理するソフトウェアである．VMM を用いて別の VM や VMM 層から監視対象 OS のメモリなどを参照することで，OS 内部からでは検出が困難なマルウェアを検出できる可能性が高まる．この手法を実現する際に用いられるライブラリとして，実行中の VM のメモリの内容を参照する機能等を提供するライブラリ LibVMI [2] がある．しかし LibVMI を利用して対象 OS を監視するには，LibVMI が提供するライブラリ関数を直接利用したプログラムを監視内容ごとに作成する必要がある．また，対象 OS 固有の構造体メンバのオフセット等のメモリ配置を強く意識した細かな計算処理などを書く必要がありプログラムが煩雑になるため，マルウェアを検出したい場合，ユーザに大きな負担がかかる．さらに，C 言語を用いてこのようなプログラムを記述した場合，コンパイルしてから実行する必要があり，ターンアラウンド時間が長いという問題点もある．

このような問題点を解決するため，本論文では，対話的に VM 内部情報を取得可能なシステムを提案する．VM 内部情報取得のための専用の言語 (DSL) を提供することで，上記のような煩雑なプログラミング作業は不要になる．また，対話的に利用することができるため，即時性が高く必要に応じた VM 内部検査が可能となる．

2. Virtual Machine Introspention

Virtual Machine Introspection (VMI) とは，動作中の VM 内部の状態を VMM や別の VM から監視するために提案された技術である．マルウェアの侵入検知システムやマルウェア解析システムなどのセキュリティシステムを実装するために利用されている [3][4].

VMI による対象 VM 内部情報の取得方法は，対象 VM 内部で行う手法と対象 VM 外部で行う手法の 2 つに大別す

[1] 電気通信大学
[a)] a1831004@edu.cc.uec.ac.jp
[b)] iwasaki@cs.uec.ac.jp

ⓒ 2019 Information Processing Society of Japan

ることができる．

対象 VM 内部で行う手法では通常，OS 内部機能を利用して必要な情報を取得し，ネットワークインターフェイスまたはシリアルポートなどの仮想デバイスを介して VMI アプリケーションに取得した情報を送信する．この手法は，後述する対象 VM 外部で行う手法に比べてオーバーヘッドが小さい，後述するセマンティックギャップの問題がない，といった利点があるが，OS 内部の機能に依存するため，例えば OS が操作不能な状態で異常停止した場合や，マルウェアによって内部機能が改竄されている場合に，内部情報を正確に取得することができないという欠点がある．

対象 VM 外部で行う手法は，対象 OS 内部機能は利用せず，VMM を通して対象 VM に関するハードウェア情報を取得する方法である．この手法では，VMM から直接参照できる情報はメモリやレジスタの内容などの低レベルな情報に限定されており，これらの情報からセキュリティシステムの監視に有用な情報に変換する必要があるという問題点がある．この問題はセマンティックギャップ問題と呼ばれており，これを解決するために，LibVMI やその他の研究によっての解決策が提案されている [5][6][7].

LibVMI とは, VM 内部情報へのアクセスを行うために設計された, VMI 用ライブラリである．ここで, VMM 環境において LibVMI を利用し対象 VM 上で作動している Linux カーネルのメモリを直接参照することで，プロセス情報を取得する例を示す．Linux ではプロセスは `task_struct` と呼ばれる構造体によって定義されており，各構造体が双方向リストとして連結されている．各構造体はプロセス ID を格納する `pid` メンバやプロセス名を格納する `name` メンバ，次の構造体のポインタを格納する `next` メンバなどの情報を保持している．リストの先頭アドレスは `init_task` と呼ばれる大域変数に保存されている．したがって，`init_task` を参照することで先頭のプロセス構造体を取得し，連結リストを辿り各構造体が持つプロセス情報からプロセス一覧を取得することができる．

LibVMI を用いて対象 VM のプロセス一覧を取得するコードの一部を図 1 に示す．このコードでは，構造体メンバの構造体先頭からのオフセットを取得するために，`vmi_get_offset` 関数を呼ぶ．その後，プロセス構造体のアドレスに取得したオフセットを加えた後で，アドレスの値を読み取る `vmi_read_addr_va` 関数や `vmi_read_str_va` 関数などを呼んでいる．以上のように，LibVMI のライブラリ関数を直接利用してプロセス ID やプロセス名など特定の情報を取得する場合，適切なオフセットを取得する関数を呼び，そのオフセット値を用いて構造体メンバの型に応じた適切なライブラリ関数を呼ぶなどの，細かな処理を書く必要がある．この作業は煩雑で間違えやすく，ユーザに大きな負担がかかるという欠点がある．マルウェアが侵入したと疑われるような事態が発生した場合には，原因を特定するために即座に内部情報が取得できるような環境が求められる．

```
vmi_get_offset("linux_tasks", &task_offset
    ,...);
vmi_get_offset("linux_name", &name_offset,...)
    ;
vmi_get_offset("linux_pid", &pid_offset,...);
vmi_translate_ksym2v("init_task", &list_head);
current_process = list_head;
vmi_read_addr_va(current_process+task_offset,
                 &next_process...);
while(1){
  vmi_read_addr_va(current_process+pid_offset,
                   &pid...);
  vmi_read_str_va(current_process+name_offset,
                  &name...);
  printf("%d %促sn", pid, name);
  ...
}
```

図 1 LibVMI を利用したプロセス一覧取得プログラムの一部

3. 関連研究

kgdb [8] は，Linux カーネルのソースレベルのデバッガとして開発されたツールである．Linux カーネルをデバッグするために，GNU プロジェクトが開発したデバッガである gdb [9] を利用しており，通常のアプリケーションをデバッグするようにカーネルデバッグを行うことができる．gdb には遠隔モードがあり，gdb が動作するマシンとは別のマシンで動作するデバッグ対象のプログラムを，シリアル通信を介してデバッグすることができる．kgdb は gdb の遠隔モードを利用することで，対象マシンのメモリを検査し，カーネル内の特定の変数の値などを取得する．kgdb を利用して VMI を行うためには，対象 OS のカーネルソースに kgdb パッチを適用してソースを修正する必要がある．また，kgdb は対象 OS のカーネル内部機能を利用して内部情報を取得しているため，マルウェアに感染しカーネル内部機能を改竄されている場合，正確な情報を取得できない可能性があるという問題点がある．本システムは，対象 OS 外部から内部情報を取得するため，マルウェアによって対象 OS のカーネル内部機能が改竄されていても，正確に情報を取得することができる．

LibVMI は，VMM の一種である Xen [10] のための VMI ライブラリとして開発された XenAccess [11] の後継の VMI ライブラリである．Xen や KVM など複数の VMM 環境で動作するように設計されており，Windows や Linux が動作する対象 VM のメモリの状態を監視するために利用される．対象 OS の仮想アドレスと物理アドレスの変換処理などがライブラリ化されており，特定のアドレスのメ

ⓒ 2019 Information Processing Society of Japan

モリ内容を取得する機能や，特定の VM の一時停止，再開機能などが提供されている．これらの機能を利用することで，実行中の対象 OS のプロセスリストやモジュールリストなどを取得することができるが，対象 OS カーネルのシンボル情報や構造体のメンバオフセット情報を予め取得しておく必要がある．Linux に関しては，対象 OS の実行状態取得のために，対象 OS 内のカーネルシンボルテーブル System.map を VMI アプリケーションが動作する監視 OS にコピーする必要がある．加えて，対象 OS 内で `linux/sched.h` から構造体のオフセット情報を取得した後，その情報を監視 OS 内のファイルに記載しておく必要がある．その後，LibVMI を利用した VMI アプリケーション内に，カーネルシンボルテーブルからのアドレスの取得処理や，オフセット情報の取得及び計算処理をユーザが記述することで，対象 OS の内部情報を取得することができる．本システムでは，LibVMI のメモリアクセス API を利用するが，対象 OS 内のカーネルシンボルテーブルをコピーする必要はなく，またユーザがオフセット情報の取得及び計算処理を記述する必要もない．

PyVMI [12] は，LibVMI の Python ラッパーライブラリである．VMI アプリケーション作成の高速化と Python で書かれることが多い他のメモリ分析ツール [13] の機能を組み込みやすくするために開発されたもので，VMI アプリケーションを，C 言語より少ないコード量で記述することができる．しかし上述した通り，カーネルシンボルテーブルからのアドレス取得やオフセット情報の取得及び計算処理は，C 言語の場合と同様にして記述する必要がある．

XScope [14] は，LibVMI を利用した VMM 環境でのルートキット検出システムである．対象 OS 内部でのプロセスリストを取得するために，対象 OS 内部で ps コマンドを実行し，その結果を VMI アプリケーションに送信することで，(内部) プロセスリストを取得する．さらに対象 OS 外部からプロセスリストを取得するために，プロセス情報が置かれる anonymous page を識別し各プロセス情報を取得することで，(外部) プロセスリストを取得している．これら二つのプロセスリストを比較することで，隠蔽された怪しいプロセスを検出する．

4. 設計

本節では，本論文で提案するシステムの設計を述べる．VMI プログラムを作成する際のターンアラウンド時間を短縮するために，対話的なプログラム実行環境とそのための DSL を設計する．

現状では，構文解析の容易さから，DSL の式は Lisp 風の S 式で与える．ユーザが記述しやすいような C 言語に似た構文の検討は，今後の課題である．対話的なプログラム実行環境は，ユーザが入力した DSL の式を読み込み，実行環境の中で評価し，評価結果を出力する，ということを繰り返す REPL (Read-Eval-Print Loop) 機構で実現する．式の評価結果は，値とその値の型 (以後，この組を「評価値」と呼ぶ) とする．評価値には型情報が含まれるため，ユーザは DSL の式において型を陽に指定する必要はない．さらに構造体型に関しては，構造体のメンバ名と型，先頭からのオフセットなどに関する情報を本システムが内部的に保持するようにし，ユーザはがオフセットを与える必要なくメンバ名だけで必要な情報にアクセスできるようにする．

たとえば，カーネル内の大域変数名をそのまま与えれば，その変数の値と型が得られる．また，ポインタ型や配列型の大域変数名を与えれば，その変数に格納されるアドレスと型 (ポインタ/配列) が得られる．構造体の大域変数を与えた場合は，その構造体変数に格納されたアドレスと構造体名で表された型が得られる．構造体のメンバの値を得る場合は，ユーザがオフセットの計算処理を記述する必要はなく，構造体にメンバ名を与えるだけで，メンバの値と型を得ることができる．REPL での評価値の出力においては，値と型を「`::`」で区切って出力する．

本システムは，ユーザの記述と理解のしやすさを支援するため，以下のようなユーティリティ関数を提供する．

- `(. s mem)`: 構造体 s のメンバ mem を取得して返す．
- `(-> p mem)`: 構造体へのポインタ p の指す先のメンバ mem を取得して返す．
- `(deref p)`: ポインタ p の指す先を取得して返す．
- `(printstring s)`: `char` 型の配列あるいは `char*`型の値 s を，終端のヌル文字まで文字列として出力する．
- `(printarray a size)`: 配列 a とサイズ `size` を指定し，a の要素を先頭から `size` 個出力する．
- `(printstruct s)`: 構造体 s の各メンバの値と型を出力する．

評価値には値と型が含まれるため，「`.`」と「`->`」の使用においては，構造体の型，メンバのオフセット値を指定する必要がない．

提案する DSL では，式の評価値を REPL 内変数に保持することができる．名前だけから REPL 内変数とカーネル内の大域変数を区別できるようにするため，REPL 内変数名は「`$`」で始まるものとする．REPL 内変数への値の初期化は `(define var e)` で行う．ここで `var` は REPL 内変数，e は式であり，式 e の評価値を `var` に設定する．さらに，REPL 内変数への値の (再) 代入は，`(set var e)` で行う．ここで `var` と `e` の意味は，`define` の場合と同じである．

DSL は制御構造としては，条件分岐を行う `if` 式，繰返しを行う `while` 式，繰返し処理からの脱出を行う `break` 文を提供する．

本論文で提案するシステムの動作例を図 2 に示す．図 2 の「`repl >`」に続いて記述されている部分，ユーザからの

```
$ ./repl
repl > k_num
1 :: int

repl > k_str
0xffffffff81e11500 :: char*

repl > (-> k_listp val)
36 :: long

repl > (printstring k_str)
Hello :: char*

repl > (printarray k_arr 4)
k_arr[0] : 3 :: int
k_arr[1] : 1 :: int
k_arr[2] : 4 :: int
k_arr[3] : 1 :: int

repl > (printstruct (deref k_listp))
id : 1 :: long
val : 36 :: long
name : 0xffffffff81a00180 :: char*
next : 0xffffffff81d7ec10 :: struct data*

repl > (define $x (-> k_listp val))
repl > $x
36 :: long

repl > (if
     >   (< 3 5)
     >   (-> k_listp id)
     >   (-> k_listp val)
     > )
1 :: long

repl > (define $list l_listp)
0xffffffff81d68570 :: struct data*
repl > (while
     >    (!= $list NULL)
     >    (print (-> $list val))
     >    (set $list (-> $list next))
     > )
36 :: long
45 :: long
54 :: long
 .
 .
 .
```

図 2 REPL の動作例

入力である．

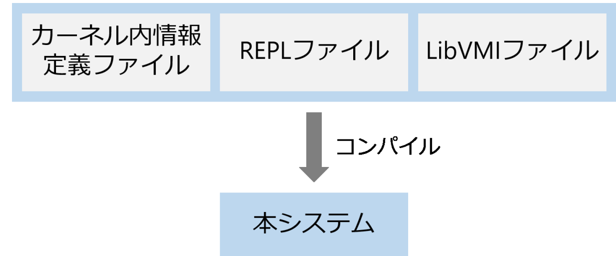

図 3 本システムの概要図

5. 実装

5.1 概要

本システムは，C 言語で記述された以下の 3 つのファイルにより構成される．概要の全体図を図 3 に示す．

- カーネル情報を定義したファイル
- REPL 処理を記述したファイル
- LibVMI の API を定義したファイル

5.2 カーネル内情報の定義

REPL に入力された式を正しく解釈し，LibVMI のライブラリ関数を正しく選択して呼び出すことを可能とするため，対象 OS のカーネル内大域変数名やそのアドレス，型情報などのカーネル内部情報を予め取得して定義する必要がある，取得したカーネル内部情報は，主に次の処理で用いる．

- カーネル内大域変数の識別 (アドレスと型の取得)
- 構造体のメンバ名の識別 (オフセットと型の取得)
- 型に応じた適切なライブラリ関数の選択

カーネル内情報定義ファイル生成の流れを図 4 に示す．対象 OS のカーネル内部情報は，デバッグ情報が付与された `vmlinux` ファイルから抽出する．ここで `vmlinux` とは，Linux カーネルの ELF フォーマットのオブジェクトファイルであり，カーネルのビルドが正常に完了すると，カーネルソースコードのトップディレクトリに生成される．`vmlinux` にデバッグ情報を付与するためには，カーネルビルド時に gcc コンパイラのデバッグ情報を付与するオプションを追加する．

本システムで利用するカーネル内部情報を作成するため，`vmlinux` からデバッグ情報のみを抽出する．Linux には，ELF フォーマットのオブジェクトファイルの内部情報を出力するツールである `eu-readelf` コマンドが備わっており，「`-winfo`」オプションを追加することでデバッグ情報のみを出力することができる．これを用いて `vmlinux` のデバッグ情報を抽出し，別ファイルに保存しておく．本システムは，このデバッグ情報から必要な情報を抽出・加工し，カーネル内部情報として C 言語で記述されたファイルを作成する．ここで，全てのカーネル内部情報を抽出すると，C 言語のファイルが巨大になってしまうため，必要

な一部の情報だけを抽出することにした．現状では，ユーザに参照される可能性の高い変数とそれに関連する構造体情報の抽出を考えているが，取捨選択の基準を適切に定めることは今後の課題としている．

5.3 REPL

REPL の全体像を図 5 に示す．

REPL を実装するために必要な機能は以下の 4 つである．

- 式を入力する部分 (Read 部)
- REPL 処理を記述したファイル (Eval 部)
- LibVMI の API を定義したファイル (Print 部)
- 以上の動作を繰り返す部分 (Loop 機構)

Read 部では，入力された式を解析し構文木を生成して Eval 部に渡す．この時，カーネル内変数名が入力されていれば，入力された変数がカーネル内情報定義ファイル内に定義されているかの検査する．Eval 部では，カーネル内変数の型情報やアドレスを取得し，LibVMI が提供するライブラリ関数を適切に選択して呼び出し，適当なバイト数のメモリにアクセスすることで，対象 OS 内データを取得して評価値とする．Print 部では，Eval 部の結果の評価値を出力する．これらの動作を繰り返すことで，式の入力，式の評価，結果の出力を連続して行えるようにする．

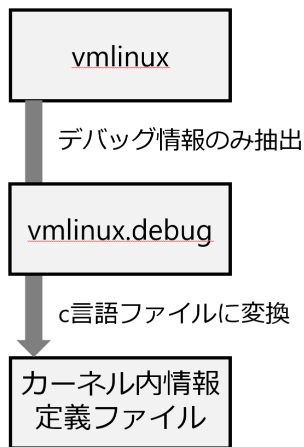

図 4 カーネル内情報定義ファイル生成の流れ

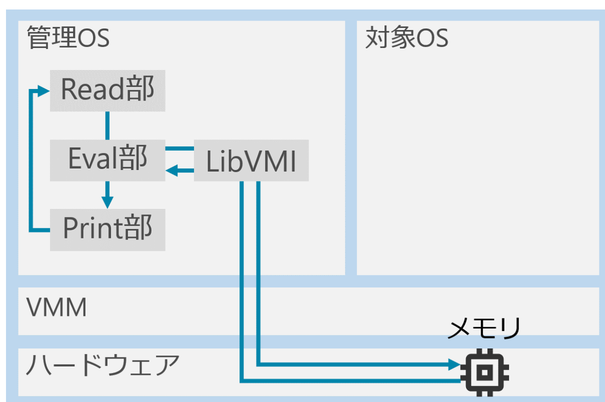

図 5 システムの全体像

6. 動作例

現状では，実装が完了してしないため，本システムの動作イメージを示す．2 章で記述したように，init_task から task_struct 構造体を辿ることで，実行中のプロセス一覧を取得することができる．本システムを用いて対象 OS のプロセス一覧を取得する動作例を図 6 に示す．

7. 課題

今後の課題として，まず本システムの実装を完了させ，本システムで実在するマルウェアを検出できるか実験を行うことが挙げられる．この実験で，正しく検出できるか，また従来のシステムより効率よく検出できるか評価することを考えている．

また，現状では DSL に必要最低限の機能のみ実装を行っているが，ユーザが本システムを利用する際，より便利に利用できるような DSL の機能の拡充を考えている．現状では，例えば関数定義の機能の追加を考えている．

また，カーネル内部情報の取捨選択の基準の設定も今後の課題として挙げられる．現状ではユーザに参照される可能性の高い変数とそれに関連する構造体情報の抽出を考えているが，より適切な取捨選択の基準を定めたいと考えている．

また，現状では本システムはマルウェアの検出のみを想定しているが，マルウェアの除去の実現も考えてる．特に，

```
$ ./repl
repl > (define $task init_task)
0xffffffff81e11500 :: struct task_struct

repl > (while 1
     >    (print (-> $task pid) (-> $task comm
        ))
     >    (set $task (. (-> $task task) next))
     >    (if (= $task init_task) (break))
     > )
  0 swapper/0
  1 systemd
  2 kthreadd
  3 ksoftirqd/0
  4 kworker/0:0H
  5 rcu_sched
  6 rcu_bh
  7 migration/0
  8 watchdog/0
  9 kdevtmpfs
  .
  .
  .
```

図 6 本システム動作例

ⓒ 2019 Information Processing Society of Japan

マルウェアプロセスを検出した際，プロセスの停止も行いたいと考えているが，Linux カーネルに元々備わっている do_send_sig_info 関数によって実現できる可能性があるため，今後，実際に使用することで調査していきたいと考えてる．

8. おわりに

本研究では，対話的に VM 内部情報を取得可能なシステムを提案した．このシステムでは，VM 内部情報取得のための DSL を提供するため，対象 OS 固有の構造体メンバのオフセット等のメモリ配置を強く意識した細かな計算処理などを書く必要がなく，VMI プログラム作成のための煩雑なプログラミング作業が不要となっている．また，対話的に利用することができるため，VMI プログラムを記述した場合，コンパイルしてから実行する必要がなく，即時性の高い必要に応じた VM 内部検査が可能となっている．

参考文献

[1] ELF_XORDDOS.AR. https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/elf_xorddos.ar
[2] Bryan D. Payne, M. Carbone, W. Lee, ”Secure and Flexible Monitoring of Virtual Machines”, Proceedings of Computer Security Applications Conference, 2007, pp. 385-397,
[3] Yacine Hebbal, Sylvie Laniepce, Jean-Marc Menaud, ”Hidden Process Detection using Kernel Functions Instrumentation”, 2017 IEEE Conference on Dependable and Secure Computing, 2017, pp. 138-145,
[4] Tamas K. Lengyel,Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, Aggelos Kiayias. ”Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System”, Proceedings of the 30th Annual Computer Security Applications Conference, 2014, pp. 386-395,
[5] Weizhong Qiang, Gongping Xu, Weiqi Dai, Deqing Zou, Hai Jin, ”CloudVMI: A Cloud-Oriented Writable Virtual Machine Introspection”, IEEE Access (Volume:5), 2017, pp. 21962-21976,
[6] Yangchun Fu, Zhiqiang Lin, ”Exterior: Using a dual-VM based external shell for guest-os introspection, configuration, and recovery”, ACM SIGPLAN Notices - VEE ’13, 2013, pp. 97-110,
[7] Yangchun Fu, Zhiqiang Lin, ”Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection”, 2012 IEEE Symposium on Security and Privacy, 2012, pp. 586-600,
[8] kgdb, https://mirrors.edge.kernel.org/pub/linux/kernel/people/jwessel/kdb/
[9] gdb. http://www.gnu.org/software/gdb/
[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, ”Xen and the art of virtualization”, Proceedings of Association for Computing Machinery Symposium on Operating Systems Principles, 2003, pp. 164-177.
[11] B. D. Payne, M. D. P. De A. Carbone, and W. Lee, ”Secure and flexible monitoring of virtual machines”, in Proc. 23rd Annu. Comput. Secur. Appl. Conf, 2007, pp. 385-397.
[12] Bryan D. Payne, ”Simplifying Virtual Machine Introspection Using LibVMI”, SANDIA REPORT(2012), 2012,
[13] Volatility. https://www.volatilityfoundation.org/
[14] Lei Cui, Zheng Song, Yongnan Li, Zhiyu Hao, ”XScope: Memory Introspection based Malicious Application Detection”, 2018 5th International Conference on Information Science and Control Engineering, 2018,

ⓒ 2019 Information Processing Society of Japan