監視対象システムを止めずに対制御データルートキットを検知するシステム

徐 振宇[1,a)]　岩崎 英哉[1,b)]

概要：対制御データルートキットとは，ユーザプロセスが発行するシステムコールのカーネル内処理ルーチンを改竄し，目的を実現するマルウェアである．この種のルートキットは，実現しやすい，汎用性が高い，そして検知されにくいという特徴を持つ．ルートキットにより汚染されたシステムの上で動作するルートキット検知システムの挙動は信用できないため，検知システムは仮想マシンモニタを利用するなどして，対象システムの外部に置くのが一般的である．しかし，従来の多くの研究では，一時的に対象システムを止めて，システムから必要な情報を取得するが，オーバーヘッドが大きく，対象システムのパフォーマンスを低下させるという問題点があった．本研究は，対象システムを止めることなく，監視対象の外部からルートキットを検知するシステムを目指す．そのため，対象システムのカーネル関数のコールフローを取得するカーネル組み込みの Ftrace を拡張し，トレース情報を対象システムの外部で取得して，ルートキットの検知に利用する．

キーワード：ルートキット検知，KVM 仮想マシン，Ftrace

1. はじめに

ルートキットとは，攻撃者が対象システムに対し，目的に応じたプログラムを埋め込むためのマルウエアである．ルートキットの特徴は，対象システムに潜伏し，攻撃者が自らの攻撃をしやすくするようにバックドアを構築し，自らの存在を隠すことである．

対制御データルートキットとは，ユーザプロセスが発行するシステムコールのカーネル内処理ルーチンを改竄し，目的を実現するルートキットである．対制御データルートキットは，実現しやすい，汎用性が高い，そして検知されにくいという特徴を持つため，もっともよく用いられる攻撃の手法である [1]．本研究は対制御データルートキット (以下単にルートキットと呼ぶ) を検知の対象として，検知システムを設計する．本システムは xftrace と名付ける．

図 1 に `read` システムコールをターゲットとした攻撃の例を示し，これを用いてルートキットの攻撃対象について説明する．この例では，攻撃者はカーネル空間内のシステムコールテーブルにある `read` システムコールのカーネル内処理ルーチンである `sys_read` 関数の場所のアドレスを書き換えることによって，攻撃を仕掛けている．ユーザプロセスが `read` システムコールを発行すると，割り込み命令によりカーネル空間のシステムコールディスバッチャに制御が移る．システムコールディスバッチャは，発行されたシステムコールの番

[1] 電気通信大学大学院情報理工学研究科
[a)] xu@ipl.cs.uec.ac.jp
[b)] iwasaki@cs.uec.ac.jp

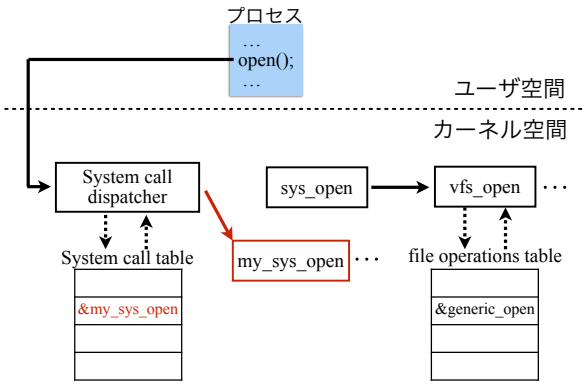

図 1: 対制御データルートキットの例

号を添字としてシステムコールテーブルを参照する．システムコールテーブルには各システムコールに対するカーネル内処理ルーチンの先頭アドレスが入っており，`read` の場合はカーネル内関数 `sys_read` の先頭アドレスになっている．図 1 では，攻撃者がこれを自身で用意した攻撃用の関数 `my_sys_read` の先頭アドレスに書き換えている．そのため，攻撃者が用意したこの関数が実行されてしまう．攻撃者がこのような手段を用いて，対象システム内にあるファイルやディレクトリなどの情報を隠蔽することができる．

また，システムコールテーブル以外にも狙われやすい標的がある．たとえば，SucKIT [7] ルートキットは Interrupt Descriptor Table を標的とし，Adore-ng [2] ルートキットは file operations table を標的とする．

ルートキットを検知するシステムのアプローチには一般的に以下の 3 つがある．

1 つ目は，マルウェアのシグネチャを利用するアプローチである．このアプローチは，対象システム上にあるファイルを全部スキャンして，マルウェアのコードに特有のバイト列が含まれているどうかを判別する．多くのアンチウィルスソフトウェアは，このアプローチを採用している．しかし，対象システムがカーネルレベルのルートキットに既に汚染され，検知システムが対象システム上で動作する場合は，検知システムが無効化される恐れがある．また，サイバーセキュリティで有名なファイルレス攻撃 [3] の検知は，このアプローチで検知できない．なぜなら，ファイルレス攻撃では，対象システムのメモリ上にあり，マルウェアの実体がファイルとして存在しないからである．

2 つ目は，差異に基づく検知アプローチである．このアプローチは，対象システム上にあるバイナリを，事前に入手した信頼できるバイナリと比較することにより，ルートキットを検知する．対象システムごとに対象システムとほぼ同様な大きさを持つバックアップを持つ必要がある．

3 つ目は，挙動ベースの検知アプローチである．このアプローチ [15] は，システムの振る舞いを観察し，ルートキットに汚染されていない時と異なる振る舞いを検知することによって，ルートキットの存在を推測する方法である．この方法には，上記の方法のような弱点がない，また対象システムに与える影響が少ないため，本研究はこのアプローチを採用する．さらに本研究では，汚染されている可能性のある対象システムから検知システムを隔離するため，仮想マシンを用いて，対象システムの外部から検知する方法を用いる．

仮想マシンを用いると，検知システムは対象システムの外にいるため，対象システムを一時的に止めて，検知システムが必要な情報を取得する方法が一般的に用いられる．しかし，対象システムを止める必要があるため，対象システムのパフォーマンスを低下させてしまう．本研究では，対象システムを止めず，外部から対制御データルートキットを検知するシステムの実現を目指す．

本研究の方針は以下の 2 つとする．1 つ目は，Linux KVM [6] 仮想マシンを用いることである．対象システムをゲスト OS とし，検知システムをホスト側で実現する．2 つ目は，ユーザプロセスによるシステムコールの発行時に，カーネル内の関数呼び出し履歴を収集することである．その理由は，3.2 節で述べるように，カーネル内の関数呼び出し履歴を分析することは，ルートキットの発見に有効であることが知られている [15], [16], ? からである．本研究では，この目的を達成するため，Ftrace [4] と QEMU-KVM [5] 及びゲスト側のカーネルを拡張し利用する．

本論文の構成は次のとおりである．まず，第 2

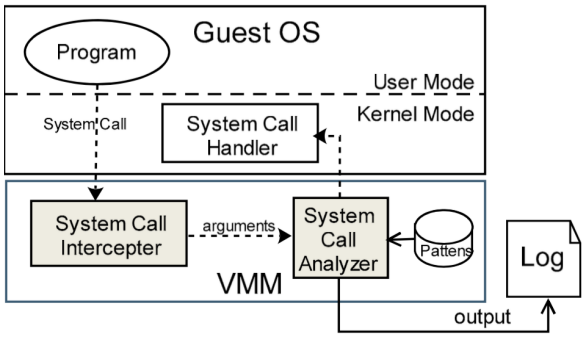

図 2: VSyscall システムの設計

節で既存システムの問題点を述べる．第 3 節で，Ftrace の説明と Ftrace を用いる理由を述べる．第 4 節では，Ftrace の問題点やシステムの設計を説明，第 5 節で本システム実装方法とユーザの利用方法を述べる．第 6 節で関連研究を述べ，第 7 節で節で本研究のまとめと今後の課題について述べる．

2. 既存システムの問題点

本節では，既存システムの例を一つ挙げて，その概要と問題点を説明する．

2.1 従来の方式

仮想マシンを用いて，検知システムを対象システムの外に設置すると，検知システムの性能が制限され，セマンティックギャップ [8] 問題が生じる．セマンティックギャップ問題とは，検知システムが直接アクセスできる CPU レジスタやメモリ内容などの低いハードウェアレベルの状態と，ゲスト OS のプロセスやカーネル内で発生したイベンドなどの高いレベルの情報との間にはギャップがあるという問題である．したがって，得られる低レベルの情報から高レベルの情報を推測しなければならない．検知システムが対象システムから情報を得るため一般的に用いられる手法は，対象システムを一時的に止めて，必要な情報をとることである．

これを，VSyscall [9] を例として説明する．ゲスト OS 内のプロセスにより呼び出されたシステムコールは，仮想マシンモニタ (VMM) に検知される．VMM は，ゲスト OS の実行状態を直接見ることができないため，ゲスト OS のシステムコールの実行を識別する．その後，VSyscall はゲスト OS から呼び出されたシステムコールイベントを解釈し，システムコールがどのプロセスに属しているかを調べ，システムコールとの関係を確立する．これにより，プロセスの動作を監視する．

図 2 に示す通り，VSyscall には，System Call Interpreter (SCI) と System Call Analyzer (SCA) の 2 つ重要なコンポーネントがある．SCI は，ゲスト OS のユーザモードによって実行されたシステムコール命令 (int 80h または sysenter) を検知し，システムコールの引数を取得して SCA に渡す．SCA はこれらの情報からシステムコール間の関係を確立し，与えられた正常時パターンに基づいてプログラムの動作を監視する．システムコールシーケンスとパターンマッチング結果はログに出力し，ルートキットの分析に用いる．

このシステムの欠点は SCI にある．SCI は，制御をゲスト OS から VMM に移すため，システムコールが呼び出される時，カーネル内処理ルーチン関数のアドレスを存在しないアドレスに置き換え，ページフォールトを起こす．このようにして，対象システムを止める必要があるため，対象システムの性能が損なわれてしまう．

VSyscall だけでなく，同様の方法を採用している研究は他にも複数ある．

VMscope [10] は，ゲスト側で動くプロセスとそのプロセスが呼び出したシステムコールの関連を調べるため，対象システムを一時的に止めてシステムコールの実行時にレジスタにある値を得る．

NumChecker [11] は，ハードウェアパフォーマンスカウンタを利用し，システムコール実行時に実際に行った機械語命令をカウントする方法を用いる．この研究では，システムコール実行前と実行後の CPU カウントにある情報をゲストのカーネルから取り出すために，対象システムを一時的に止める必要がある．

このように，ゲストからホストにある検知システムに情報を送るため，対象システムを一時的に止める方法が一般的である．これに対して，本研

究では対象システムを止めないため，外部から対象システムのメモリを直接訪問することにより，システムを実現する．

2.2 仮想マシンを止めないシステム

LibVMI [12] は，Xen [13] 仮想マシンを対象として設計された Vitrual Machine Introspection (VMI) を行うためのツールであり，これを用いれば，ゲスト OS (ドメイン U) を止めずにゲスト OS の情報を得ることができる．例えば，ゲストが発行したハードディスクに対する I/O 処理の監視や，ネットワークの監視などの機能を提供している．

LibVMI は，ホスト側 (ドメイン 0) の `xc_map_foreign_range` 関数を用いて，検知システムが欲しい情報を，ドメイン U から共有メモリ上に持ってくる．[*1] しかし，共有するメモリ量が限られているため，一度に持って来ることのできる情報の量が制限されている．このため，情報が大きいと，`xc_map_foreign_range` 関数の利用頻度が高くなり，コストがより大きくなるという欠点がある．ルートキットの存在を検知するためには，ゲスト OS から何度も情報を取得する必要があるため，コストが更に大きくなる．

3. Ftrace

本節では，Ftrace の用途及び基本的な使い方を述べる．そして，実例を用いて，本研究で Ftrace の利用方法及び Ftrace の有用性について説明する．

3.1 基本的な仕組み

Ftrace は Linux カーネルに組み込まれているトレース機構である．その特徴は，動作中のカーネル内の関数呼び出しのイベントを記録し，ログとして出力することである．一般的に，カーネル内の関数の待ち時間やパフォーマンスの問題のデバッグや分析に使われている．

図 3 に示すように，Ftrace は以下の 4 つの部品で構成されている．

[*1] この関数の中で，実質 `mmap` を呼んでいる．

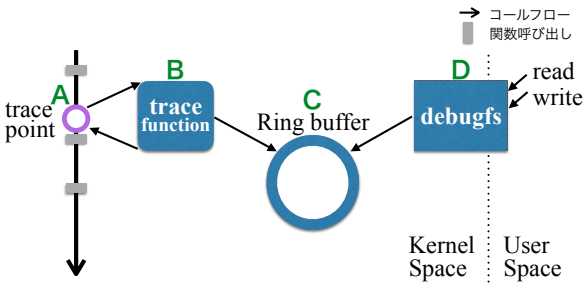

図 3: Ftrace の構造図

トレースポイント

トレースポイントは図 3 中の A に示した場所にある．その役割は，トレースしたい関数を実行する前に，トレース関数を先に呼び出すことである．トレースポイントは次の 2 つの処理で実現されている．1 つ目は，カーネルビルド時に，カーネル内のすべての関数定義の先頭にトレース関数を呼び出すためのアセンブリコードを埋め込み，更に，その場所を記録することである．ただし，この時に埋め込まれたコードは，カーネルイメージを作る時には，`nop` として書き換えられて，無効化されている．このようにすることで，デフォルトの状態ではトレース関数が呼び出されず，性能が損なわれない．2 つ目は，実際にトレースを行いたい時には，ユーザがトレースしたい関数にある `nop` を，トレース関数を呼び出すためのコードに書き換えることである．[*2]

トレース関数

トレース関数は図 3 の B にある．その役割は，ユーザが指定したトレースの機能を果たして，トレースの結果をリングバッファに送信することである．

リングバッファ

リングバッファは図 3 の C にある．その役割は，トレース結果を保存することである．カーネルのリングバッファは，Ftrace だけでなく，様々なログ出力機構に使われている．

デバッグファイルシステム

図 3 の D に該当する．その役割は，ユーザはこの機構を利用して，リングバッファに保存した情

[*2] Ftrace はブレイクポイントを用いて，コードの書き換えを行っている．

報を出力することである．

3.2 Ftrace の有用性

Ftrace の利用方法を，次の例を用いて説明する．この例では，open システムコールに与えられた引数を出力するように sys_open をフックしている．

カーネルが正常な時の Ftrace の結果を図 4(a) に，システムコールテーブルが書き換えられている時の Ftrace の結果を図 4(b) に示す．

二つのコールフローの結果を比較すると，open システムコールが怪しいのが分かる．図 4(b) では，本来の open システムコールによるカーネル内関数 sys_open の呼び出しが，アドレスが 0xffffffffc0b37000 である関数の呼び出しに置き換えられており，システムコールテーブルが汚染されている可能性が高いと判断することができる．

更に，前で述べた通り，Ftrace の利用は対象システムを止める必要がないため，本研究では，対象システムのコールフローを取る手段として，Ftrace を用いる．

4. システムの設計

本節は xftrace の概要を示した後，まず Ftrace の利用について説明する．次に，Ftrace の利用により生じる問題点を述べる．最後に，問題の解決方法を含めた xftrace の設計を述べる．

4.1 Ftrace の利用

xftrace の概要を図 5 に示す．xftrace の設計に関する目標は，ホスト側で，仮想マシンを止めずに，ゲスト側のカーネルコールフローをホスト側で出力することである．そのため，ゲスト側の Ftrace を用いて，目的を達成する．トレース結果は，ホスト側でゲスト OS のメモリを覗くことにより直接取得する．

Ftrace は対象システムのコールフローを取得するが，対象システムの内部にあるため，トレースの結果が攻撃者により改竄される恐れがある．例えば，リングバッファにあるトレース結果を抹消し．怪しい関数呼び出しの履歴を隠すような改竄が考えられる．

4.2 攻撃者による改竄の防止

攻撃者によるリングバッファ改竄を防止するため，我々は幾つかの方法について考察した．

まず第一に，トレースの結果を直接ホストに送信するように，システムを設計する方法が考えられる．この方法の問題点は，ゲストから外に情報

```
# tracer: function_graph
#
#CPU DURATION  FUNCTION CALLS
# |   |   |     |   |   |   |
0)           |do_syscall_64() {
0)           |  syscall_trace_enter() {
0)           |    /*sys_open(filename:7f5ba2
                    516ad0,flags:0,mode:0)*/
0) 0.191 us  |  }
0)           |  SyS_open() {
0)           |    do_sys_open() {
0)           |      getname() {
0)           |        getname_flags() {
0)           |          kmem_cache_alloc() {
0) 0.045 us  |            _cond_resched();
0) 0.381 us  |          }
...
```

(a) 正常時

```
0)          |do_syscall_64() {
0)          |  syscall_trace_enter() {
0)   |     /*sys_open(filename:7ffe6072e100,
             flags:80000,mode:7ffe6072e13)*/
0) 0.521 us|  }
0)          |  0xffffffffc0b37000() {
0)          |    printk() {
0)          |      vprintk_func() {
                     ...
0) 4.293 us|           }
0)+10.759us|         }
0)+11.113us|       }
0)+11.540us|     }
0)+11.909us|   }
0)          |   SyS_open() {
0)          |     do_sys_open() {
0)          |       getname() {
0)          |         getname_flags() {
0)          |           kmem_cache_alloc(){
0) 0.050 us|             _cond_resched();
0) 0.562 us|           }
...
```

(b) システムコールテーブルが書き換えられた時

図 4: Ftrace の結果

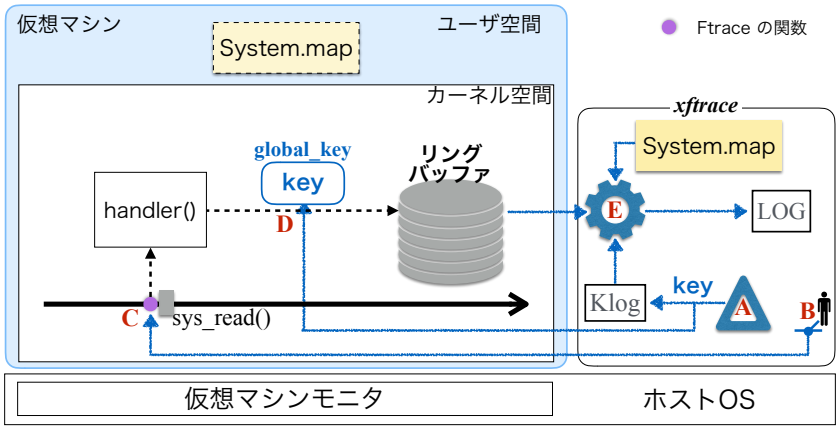

図 5: xftrace の概要

を送信するために，対象システムを一時的に止める必要があるという点である．その理由は，2 節で述べた通りである．

第二は，xftrace が動作する間に，ゲスト OS のメモリ内で新たに割り当てられた領域にトレース結果を生成するようにする方法である．たとえば，LKM を用いて，トレース結果の送信先を置き換えることにより機能の実現ができる．この方法の問題は，LKM が既に攻撃者により汚染されている恐れがあるため，トレース結果の信用性が失われることである．

最終的に本研究では，バッファオーバーフロー攻撃に対する防御方法を参考として，トレース結果をリングバッファに書き込む際に，事前に用意した鍵と排他的論理和 (XOR) をとってから書き込むように，xftrace の設計を定めた．この機能は Ftrace の機構を拡張することにより容易に実現できる．鍵は，ホスト側で一定の時間間隔で新たに生成し，その鍵を直接ゲスト OS のメモリに書き込むことにより，ゲストに反映するように，システムを設計した．

本機構は，鍵を利用するための拡張など，ゲスト OS のカーネルの拡張も必要である．ゲスト OS の内部に鍵を格納する場所を事前に確保し，ホストからゲストのメモリを直接書き換えるようにする．本システムに対し，攻撃者はトレース結果を偽装しようとしても，改竄したい関数を格納する場所はどこにあるのかを調べ，更に異常が検知されないように該当の内容を書き換えるのが難しくなる．

しかし，この方法の利用により新たに生じる問題もある．ゲスト側における鍵の読み取りと，ホスト側で新たに生成した鍵の書き込みによる競合である．この競合により，XOR に用いる鍵の値として過渡的な値が読めて，結果としてリングバッファにおかしな値が書き込まれる可能性がある．この問題点への対処法は，以下 2 つが考えられる．1 つ目は，ルートキットの判断を，一回だけのトレース結果からだけではなく，複数回トレースを分析することにより行うことである．この競合が発生する頻度は低いと考えられるため，ルートキットの検知に及ぼす影響が少ない．2 つ目は，メモリ同期処理を行うことである．鍵をゲストのメモリに書き込む前に，そのメモリに対する書き込みができるか否かの判断を行う．たとえば，QEMU-KVM 内部にある機構により実現できる．メモリ同期処理により発生するコストが大きいため，本研究では，競合の可能性は高くないと判断し，1 つ目の方法を採用する．

4.3 コールフローの構築

ホスト側でコールフローを生成するため，以下 3 つが必要である．

a ホスト側で，対象システムのメモリバッファにあるトレース結果を取得する．ホスト側にある対象システムを格納したメモリファイル

から，リングバッファにアクセスし，値を取得する．この時に，メモリファイルから取得した内容は，鍵と XOR をとった関数の仮想アドレスである．

b ホスト側で，対象システムのカーネルがビルド時に生成したシンボルテーブル System.map を保持する．ここにはカーネルの関数やデータ構造とそれらの仮想アドレスが格納されている．

c ホスト側にある配列 Klog に，鍵と時間を履歴情報として保持する．この鍵の履歴は，トレース結果の中の帰り番地を正しいものにするために利用する．

4.4 xftrace の構成

図 5 に基づき，xftrace を構成する部品について説明する．

A：Key_Controller

Key_Controller は鍵を生成するための部品であり，2 つの機能を果たしている．1 つ目は，一定時間間隔で新しい鍵を生成し，それを Klog に履歴として保存することである．2 つ目は，生成した鍵をゲスト OS 内部の global_key に書き込むことである．鍵は 64 ビットの符号なし整数値である．

B と C：Trace_Switch

Trace_Switch は，トレースの有効化・無効化を行うための部品である．ホスト側 (B) で Ftrace の tracer を `function_graph` に設定し，Ftrace (C) を有効にする．あるいは tracer を `nop` に設定し，Ftrace を無効にする．

D：Push_Raw

Push_Raw は，ゲスト側の Ftrace のトレースの結果を global_key と XOR を取って，メモリバッファに書き込む．

E：Log_Maker

Log_Maker は，4.3 節で説明した部品 a，b，c を利用して，コールフローを生成する．

5. xftrace の実装

この節で，xftrace の実装方法について述べる．まず，仮想マシンのメモリに対して直接操作する方法を述べる．次に，xftrace の各部品を実現する方法について説明する．

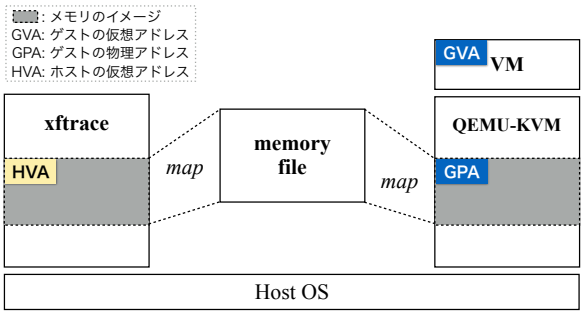

図 6: メモリ操作のイメージ図

5.1 仮想マシンへの操作

ゲストのメモリの確保

本研究は KVMonitor [14] のメモリ操作方法を参考として，ホストからゲストのメモリを直接操作する機能を実現する．

本研究は，図 6 に示す通り，VM に割り当てる物理メモリをファイルとしてホスト上で作り，そのファイルを，VM と xftrace 用のプロセス空間に，`mmap` システムコールを用いてマップするようした．メモリファイルは QEMU-KVM が作成するバッキングストアを利用した．本来の QEMU-KVM では，他のプロセスがバッキングストアを利用できないように，`unlink` を用いて，バッキングストアファイルへのリンクが削除されている．本研究は，バッキングストアを削除しないように，QEMU-KVM に変更を加えた．更に，バッキングストアを生成する時に，書き込みと読み取り権限を与えた．

さらに，仮想マシンを立ち上げる時に，事前に指定した大きさのメモリを確保するように，QEMU-KVM のオプションを指定した．これは，ホスト上で連続した仮想アドレスを仮想マシンに割り当てるためである．

これにより，QEMU-KVM により生成される VM は従来の通りに動き，我々は VM を止めずに，ホスト側で対象システムへの操作ができるようになる．

アドレスの変換

ホスト側でゲストのメモリを操作するには，ゲストの仮想アドレス (GVA) をホストの仮想アドレス (HVA) に変換する必要がある．ゲストの物理アドレス (GPA) を経由して，アドレス変換では，まず GVA から GPA を計算し，次に GPA から HVA を計算する．

GVA はカーネル空間とユーザ空間に分れているため，GVA から GPA の計算は，それぞれの空間に対して行う必要がある．カーネル空間にあるアドレスの変換は，仮想アドレスからカーネルのメモリ空間の開始仮想アドレスを引くだけで求まる．なぜなら，カーネル空間は仮想アドレス上で連続に割り当てられていて，かつ物理メモリの 0 番地から始まるためである．xftrace では，ユーザ空間に対して操作する必要がない．

GPA から HVA の計算も単純である．xftrace で，mmap システムコールを用いて，メモリファイルを VM と xftrace にマップしたため，GPA と HVA のアドレスは一対一対応になっている．このため，GPA から HVA へのアドレス変換は，マップの開始アドレスをオフセットとして，計算する．

5.2 システム各部品の実現

Key_Controller:

Key_Controller を実現するため，ゲストのカーネルにグローバル変数 global_key を作る．そして，xftrace は global_key のアドレスを持つ．それから，鍵を生成する関数 key_maker で鍵を作る．生成された鍵を直接ゲストカーネル内に用意したグローバル変数 global_key に置き，時刻と鍵をホストにある配列 Klog に出力する．

Trace_Switch:

Ftrace の tracer を function_graph に設定し，tracing_enabled [*3] の値を 1 とすることにより，トレースを開始する．また，tracer を nop に設定し，tracing_enabled の値を 0 することにより，トレースを完了する．

[*3] 対象システムのカーネル内にある Ftrace を起動するためのグローバル変数

Push_Raw:

通常の Ftrace では，ftrace_return_to_handler 関数の実行により，リングバッファにトレース結果を送る．xftrace では，我々はその関数が呼び出される時に，送信の内容と global_key の XOR を取ってから送信するようにカーネルを拡張する．

Log_Maker:

ホスト側で，直接ゲストのメモリバッファから，トレース結果を取る．そして，xftrace の関数 trace_maker により， System.map とトレースの結果と Klog の情報により，トレースの結果を生成する．

5.3 ユーザの利用方法

Ftrace の有効化/無効化の設定

xftrace を利用するため，ホスト側で xftrace コマンドを提供する．xftrace の第一引数は，トレースの開始と終了を制御する．

トレース対象の設定

ユーザがトレースしたい関数を設定する．このため，xftrace の第二引数以降の引数は，トレースしたい関数の関数名である．

ログの出力

実行結果として，コールフローを収納したファイル trace_graph.txt が生成される．

実行例

xftrace コマンドの実行例を次に示す．この例では，sys_open と sys_read をトレースの対象としている．

```
$ xftrace on sys_open sys_read
$ xftrace off
$ ls
 trace_graph.txt
```

6. 関連研究

KVMonitor [14] は，KVM 仮想マシンを対象として，ゲスト上で動く検知システムをホスト上で動かすためのツールである．検知システムをホスト側上で動かすため，ホスト側で検知システムが必要

な情報を提供する必要がある．例えば，ゲストのメモリから再現したプロセスリストの情報，ハードディスクにあるファイルの情報，ネットワークに関わる通信パケットの情報等を KVMonitor で提供できるように設計されている．本研究は，ゲストにあるカーネル関数呼び出し履歴を収集するため，ゲストのメモリを直接覗き見る機能が必要である．この点に関して，KVMonitor を参考した．

CFI [15] は，コールフローのインテグリティをチェックする研究である．CFI を用いると，ソフトウェアが乗っ取られてされているか否かの判断ができる．CFI では，監視対象の粒度の設定について，トレードオフが生じる．細かい粒度に設定すると，パフォーマンスが損なわれる．逆に大きい粒度に設定すると，検知の対象が限られて，攻撃に十分の隙間が生じる．xftrace は，カーネル関数を対象として，検知を行う時のみ，Ftrace を動作させるため，常に起動状態である CFI よりコストが小さいである．

SecVisor [16] と OSck [17] は，VMM を使ってカーネルのコールフローのインテグリティを保証する手法である．SecVisor は，セキュア OS を用いて，ユーザが意図しないカーネルへの変更を防ぐことができる．しかし，この方法では，ユーザの自由度がかなり制限されてしまうという問題が生じる．OSck は，カーネルのインテグリティを保証するため，静的にカーネルのコードを分析する手法と動的にポインタが指すオブジェクトの型を分析する手法を用いている．しかし，カーネルのコードを分析する時に，仮想マシンの情報を得るため，対象システムを止める必要がある．これに対して，本研究では，それらの問題が存在しない．

DevOps [18] は，クラウドの上で稼働するシステムの性能を劣化させることなく，VMI を行うためのシステムである．対象システムに及ぼす影響を減らすため，DevOps はライブマイグレーションを用いて，対象システムのクロンを行い，クロンされたシステムに対する VMI を行う．しかし，ライブマイグレーションを行うため，僅かの間だけ対象システムを停止させている．本研究は，異なるアプローチを用いて，対象システムを停止することなく検知を行う．

Virtio-trace [19] は，ホスト側で，仮想マシンに関連するデバッグ情報を出力する研究である．Virtio-trace はゲスト OS とホスト OS で取得したトレースデータをマージし，時間順で並べて表示することで，ゲスト OS 上で発生した問題に対して，ゲストまたはホストの原因の判断を容易にする特徴がある．しかし，ゲスト OS から取得したトレースデータはルートキットに感染した場合には信用できないため，本研究の目的に Virtio-trace を利用することはできない．

7. おわりに

本論文では，VM を止めずに，VM の外から対制御データルートキットを検知するためのシステム xftrace を提案した．本研究で，まずカーネル関数のコールフローをとることが，対制御データルートキットを検知するのに有効であることを説明した．xftrace は，QEMU-KVM のメモリファイルと拡張した Ftrace を用いて，VM を止めないでトレースを取ることを目指している．さらに，攻撃者によるトレースデータの改竄を防ぐため，VM の上で動く Ftrace の結果を用意したキーと XOR をとり，VM 外で生成するようシステムを設計した．xftrace は，従来の手法と異なり，ホスト側でゲストのプロセスリストのような情報を再構築する必要がないため，VM を止める必要がなく，パフォーマンスの期待もできる．

現状は，最低限の機能を揃えた動作するシステムを仕上げるため，簡略化バージョンのシステムを実装している．今後の課題は，本稿で述べていない部品として，Ftrace のトレース関数を監視する機構の追加も考えられる．最終的に，対象システムに何種類のルートキットを仕掛けて，それらのルートキットを xftrace で検知することにより，性能評価を行う予定である．

謝辞　研究を進めるにあたって熱心なご指導をいただきました電気通信大学の中野圭介准教授ならびに岩崎研究室，中野研究室の皆様に心から御礼申し上げます．また，KVMonitor を提供して下さった九州工業大学の光来健一准教授に感謝致し

ます．また，本論文をまとめるにあたり有益なコメントを下さった東京農工大学の山田浩史准教授，東洋大学の浅野智之助教に深く感謝いたします．

参考文献

[1] J. N. L. Petroni and M. Hicks, "Automated detection of presistent kernel control-flow attacks", in Proc. 14th ACM Conf. Comput. Commun. Secureity, Alexandria, VA, USA, Oct. 2007, pp. 103-115.

[2] May 30. 2012, The Adore-ng Rootkit, https://stealth.7350.org

[3] July. 2017, Symantec, Internet Security Threat Report, https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/istr-living-off-the-land-and-fileless-attack-techniques-en.pdf

[4] 2008, ftrace - Function Tracer, https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[5] QEMU-KVM, https://wiki.qemu.org/Features/KVM

[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, "kvm: the Linux virtualmachine monitor." In: Proceedings of the Linux symposium. vol. 1, 2007, pp. 225230

[7] Sd and Devik, "Linux on-the-fly kernel patching without LKM.", Phrack Mag., vol. 11, Jan. 2014.

[8] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, R. Sion, "Introspections on the Semantic Gap", IEEE Security & Privacy Volume: 13, Issue: 2, Mar.-Apr. 2015, pp. 48-55

[9] B. Li, J. Li, T. Wo, C. Hu, L. Zhong, "A VMM-based System Call Interposition Framework for Program Monitoring", Parallel and Distributed Systems , 2010 IEEE 16th International Conference, 8-10 Dec. 2010

[10] X. Jiang, X. Wang, "Öut-of-the-boxM̈onitoring of VM-based High-Interaction Honeypots", RAID'07 Proceedings of the 10th international conference on Recent advances in intrusion detection, Gold Goast Australia, September 05-7, 2007, pp. 198-218

[11] X. Wang, R. Karri, "Reusing Hardware Performance Counters to Detect and Identify Kernel Control-Flow Modifying Rootkits", 2016 IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 485-498

[12] B. D. Payne, M. D. P. de A. Carbone, W. Lee, "Secure and Flexible Monitoring of Virtual Machines", Computer Security Applications Conference, 10-14 Dec. 2007.

[13] P. Barham et al., "Xen and the art of virtualization", in Proc. 19th ACMSOSP, New York, NY, USA, 2003, pp. 164177.

[14] K. Kourai, K. Nakamura, "Efficient VM Introspection in KVM and Performance Comparion with Xen", 2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing, pp. 192-202

[15] M. Abadi, M. Budiu, U. Erlingsson, J. Ligatti, "Control-Flow Integrity Principles, Implementations, and Applications", ACM Transactions on Information and System Security, Volume 13 Issue 1, October 2009 Article No. 4

[16] A. Seshadri, M. Luk, N. Qu, A. Perrig, "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes", SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pp. 335-350

[17] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, E. Witchel, "Ensuring Operating System Kernel Integrity with OSck", ACM SIGARCH Computer Architecture News - ASPLOS '11, Volume 39 Issue 1, March 2011, pp. 279-290

[18] S. Suneja, R. Koller, C. Isci, E. Lara, A. Hashemi, A. Bhattacharyya, C. Amza, "Safe Inspection of Live Virtual Machines", VEE '17, Volume 52 Issue 7, July 2017, pp. 97-111

[19] virtio-trace: Support virtio-trace, https://lwn.net/Articles/508063