VMM上で動作するスクリプト言語による ゲストOSの監視ルール記述

市川 遼[1,a)]　並木 美太郎[1]

概要：マルウェア解析において，実際にプログラムを動作させて挙動を観測する動的解析は非常に有用な手段である．動的解析によって得られる情報は実際に発生するシステムコールの呼び出しやファイルへのアクセスなど，多岐に渡る．中でもとりわけ重要なのは，マルウェアが実行された後に発生するネットワークアクセスである．特に遠隔操作を行うためのマルウェアは，攻撃者の命令を受け取るために外部のサーバへ通信を行うため，このアクセスを遮断することはさらなる被害拡大の抑止につながる．そこで本研究ではOSのネットワーク通信を監視・制御するための仕組みを提案し，我々の開発したLVisorに追加実装を試みた．LVisorはマルウェアの実行を契機とするためにプロセス生成などを含むOSレベルの情報を監視できるようにし，これらをプログラマブルに扱う手段としてLuaを組み込んでいる．また高い権限で動作するマルウェアの解析妨害を回避するため，BitVisor上に実装を行なっている．本論文ではネットワーク通信へ介入するための仕組みを設計し，Luaに追加するAPIの設計，実装について述べる．

キーワード：マルウェア, 仮想計算機モニタ, 動的解析

1. はじめに

サイバー攻撃に用いられるマルウェアを検知する手法としては，パターンマッチやヒューリスティクス，ビヘイビアベースなどがある．シグネチャマッチ，ヒューリスティクスではマルウェアの実行プログラムに含まれる命令列を検知に用いるため，動的にコードを展開するマルウェアに対しては無力である．ビヘイビアベースの検知は，マルウェアの挙動を正確に捉えることができるため極めて強力であるが，OSレベルの動作を観測する必要があるため，OS内にエージェントを配置することがほとんどである．

一方で近年のサイバー攻撃で用いられるマルウェアにはカーネルの権限で動作するものがあり，OS内のエージェントは全てマルウェアに晒されることになる．この場合仮想マシン (VM) を用いたサンドボックスを用いることは必至であるが，VMを用いた解析を行う際にはセマンティックギャップを避けては通れない．セマンティックギャップとは，OSレベルの意味 (semantic) を物理デバイスレベルから解釈しようとする際に生じる隔たり (gap) のことを指す．VMからゲストOS内部の挙動を観測する際にはセマンティックギャップに直面するため，これを解決しなければ監視システムを実現することはできない．そこでこれらの問題を解決するシステムとして，LVisorを開発した [5]．

ネットワーク通信の監視を行うメリットは被害の拡大を防ぐことである．愉快犯によるランサムウェアなどを除いた多くのマルウェアは，起動したことを攻撃者に通知するため，また攻撃者の命令を受け取って実行するためにネットワーク通信を行う．この通信を制御すれば，APT (Advanced Persistent Threat) などに見られる水平展開を抑制することができ，マルウェアによる感染拡大の抑止につながる．そこでネットワーク通信の監視・介入を可能にする仕組みを本論文で提案し，我々の既存論文 [5] で提案したLVisorに追加する．

2. 先行研究

マルウェアの検知に関する手法としては，マルウェアのプログラム本体を静的に解析する手法と，実際にマルウェアを動作させてその挙動を調べる手法がある．静的解析は主にマルウェア本体に対して解析を行うのに対し，動的解析はマルウェアを実際に動作させてその挙動を解析する．

静的解析に関する先行研究としては，機械語の類似性に基づいた自動分類 [6] や，機械語のCFG (Control Flow Graph) の類似性による分類 [8], API推移グラフを抽出して分類を行う手法 [7] などがある．これらはいずれもマルウェアのプログラム本体に含まれるデータに対する解析で

[1] 東京農工大学
[a)] s177837r@st.go.tuat.ac.jp

あり，固有のパターンを抽出することで類似性を評価する．しかし静的解析はプログラムを動作させないため，実行時にコードを展開したり難読化がかかっているマルウェアに対してはあまり効果がない．

そこで動的解析ではプログラムを実際に動作させてその挙動に焦点を当てる．近年の研究では実際にマルウェアを動作させて重要情報へのファイルアクセスに注目するもの [9] があり，テイント伝搬と呼ばれる手法では実際にマルウェアを動かして興味のあるデータの流れに注目する [10]．ところが解析妨害機能を持つマルウェアのうち，例えば他のプロセスにコードを注入して追跡されないようにするものに対しては，OS 内からの解析に限界があり，これを解決すべく VMM からシステムコールを追跡する手法 [11] や，VM を用いて外側から解析を行う手法 [12][13] などがある．VM 上でゲスト OS の解析を行う場合，問題になるのは先に挙げたセマンティックギャップであるが，DECAF[12] では QEMU に動的バイナリ変換を利用してテイント解析を行い，DRAKVUF[13] では Google の Rekall[3] と Xen の API を用いて仮想アドレスの変換を行うというアプローチをとっている．

また，マルウェアの行う通信に着目した研究も広くされており [14][15]，マルウェアに感染した後に被害を抑止するための研究 [16] もある．

課題

動的解析の方がより高級な情報を得られるが，実際にマルウェアを動かすため解析妨害機能の影響を直に受ける．解析妨害機能を回避するためには仮想マシンを用いるが，既存の仮想マシンを用いたサンドボックス [12][13] にはネットワークを監視する機能はない．また動的にネットワークのルータを制御する研究 [16] にあるように，ネットワークの経路制御を行うメリットは大きい．ただし物理ルータを用いた経路制御は使用する機器に深く依存し，汎用的な処理を記述することができないため，これを本研究で解決すべき課題とする．

3. 目標

本研究で解決する課題に対して，ネットワーク通信の経路制御を実現することを目標とする．このためにはネットワークで制御すべき対象となる層は L2 であり，経路制御のタイミングを判断するためにゲスト OS の情報が必要になる．ネットワーク通信を L2 の層から制御するため仮想的なルーティングが可能であり，これは SDN の実現につながる．

(1) 制御すべき層

ネットワークへの介入に必要な機能は，NIC を流れるパケットの一番外側でカプセル化を行っている Ethernet ヘッダより下の操作である．このため L2 レイヤから制御を可能にする必要があり，また汎用的に用いられる IP ヘッダ，TCP/UDP ヘッダの制御も必要である．これらのヘッダを操作することで，TCP 通信のデータ監視だけでなく送信先アドレスの書き換えを行うことも可能である．これらの処理はケースに合わせて柔軟に書く必要があるため，汎用的な処理が記述できるプログラミング言語，特にコンパイル不要なスクリプト言語であることが望ましい．

(2) ゲスト OS との連携

ネットワークへの介入をする処理を記述しても，その実行タイミングを制御できる方が望ましい．実行された後にネットワークを通じて外部に情報を送信するマルウェアの場合，ゲスト OS で発生したイベントと紐付けて処理の有効化を行う必要がある．このため，ゲスト OS でプロセスを生成したことを検知してネットワーク通信の制限をかけられるようにする．

(3) SDN の実現

L2 への介入は SDN の構築に用いることも可能である．Ethernet ヘッダを編集することで，本来 L2 で接続されているデバイスに紐付けられている MAC アドレスを無視することができるため，最初の hop 先を変更可能で，この変更はゲスト OS から検知することはできない．これを複数のマシンに導入し，お互いにデータを同期することで仮想的なネットワークを構築可能であり，物理デバイスを必要としないルータを構築することができる．またこのような複雑な処理を記述するのには，同様にプログラミング言語が適している．

そこで本研究ではネットワーク上を流れるデータをスクリプト言語で処理できるようにすることを目標とする．そのために OS より低い層に位置する VMM にスクリプト言語の処理系を組み込み，ゲスト OS の通信に対して操作を行う API を提供する．同様に VMM とゲスト OS 間のセマンティックギャップを解決し，ゲスト OS 内部の処理に対してもフックを設定できるようにする API を提供する．

4. 設計

LVisor と本システムの全体設計を図 **4** に示す．LVisor に加えて，言語処理系を用いてネットワーク監視ルールを処理し，パケット処理用のプログラムを埋め込むことによってゲスト OS のネットワーク通信へ介入することを可能にしている．

LVisor の OS 監視ルールには言語処理系に実装した API

を用いている．同様に，今回新たに提案するネットワーク監視にも専用の API を設計し，ルールから用いるようにする．ネットワーク監視の目的は VMM からパケットの操作を可能にすることであるが，これには L2 だけでなく，L3 やその上のレイヤーにも介入できることが望ましい．原理的には L2 のパケットを編集することができればカプセリングを解除することで L3 などへの干渉も可能であるが，より汎用的な用途を考慮して今回は L3 (IP パケット) および TCP/UDP パケットへの干渉を行える API を設計した．

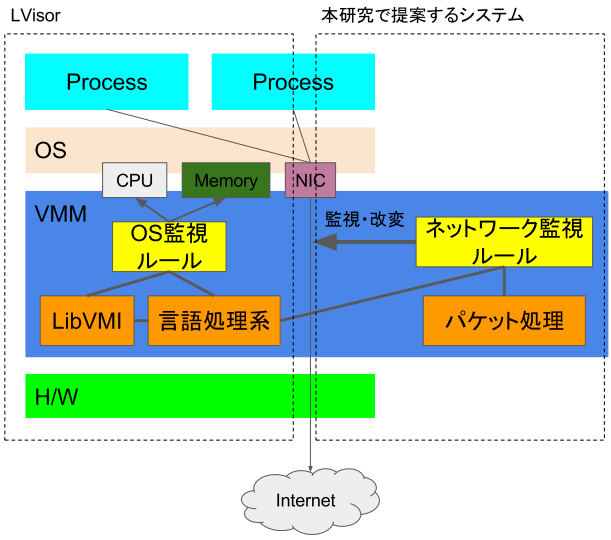

図 1　全体構成

4.1　ネットワーク監視

VMM からゲスト OS のネットワークの流れを監視するには，ゲスト OS に対して提供する NIC にフック処理を追加する．ゲスト OS が行う通信は全て NIC を介して行われるため，OS より更に下のレイヤーで処理を書くことで通信に対しての介入が自由に行える．通信への介入を行う仕組みとして，hypervisor を利用する．hypervisor 中で NIC のフックを行い，パケットを処理する.(**図 4.1**)

フック処理内で行う手続きを動的に変更することができれば，状況に応じてネットワークの通信を制限することが可能になる．

そこで，この手続きの内容をプログラミング言語で記述することを考える．ファイアウォールの処理をプログラマブルに記述するモチベーションは，例えば YAMAHA 社のルータに搭載されている Lua スクリプト機能などに挙げられる．異常検知に用いたり，定期的にログを送出するなどに用いることが可能であり [2]，セキュリティの目的においても同様のことを実現するメリットがある．プログラミング言語で手続きの内容を記述するためには，ネットワークの監視機能に直結する API が必要であり，この設計は 4.3 で述べる．

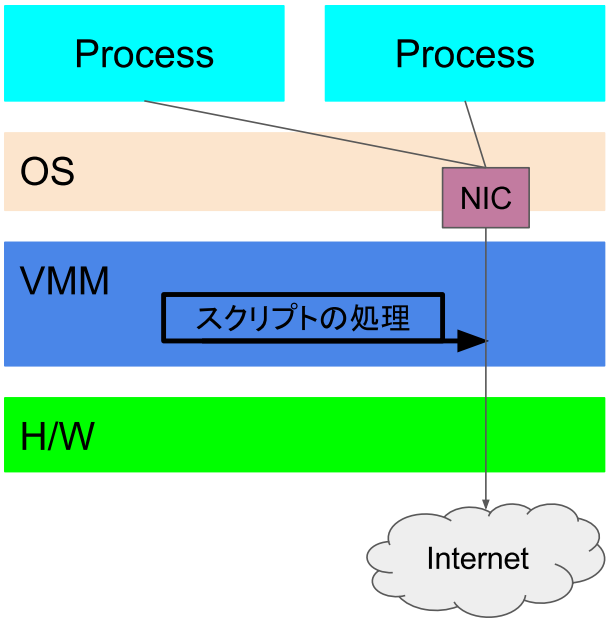

図 2　NIC のデータを VMM から処理する様子

4.2　OS レベルの監視

ネットワークの監視機能を用いるタイミングはその目的によって決定されるが，マルウェア検知においてとりわけ重要なのは疑わしい処理の実行である．この処理を検知するためには VM と OS 間のセマンティックギャップを解決する必要があり，本研究では LibVMI[4] を用いた．しかし LibVMI はあくまで物理メモリと仮想メモリの変換を補助するライブラリであるため，実際にゲスト OS の処理をフックするためには常に VMM で監視を行い，LibVMI で変換した結果に応じて処理を書く必要がある．ゲスト OS の全てのステップに処理を挟むのは現実的ではないため，CPU レジスタアクセス，メモリアクセスに焦点を当てる．CPU レジスタ，メモリへのアクセスを監視する処理を登録すると VMM 側で監視対象のアドレスに対してフックを行う．フックの対象となるアクセスが発生した場合は登録したフックを呼び出し，対応した処理を実行する．これは VMM 側でゲスト OS へ提供している仮想 CPU および仮想メモリに処理を追加すればよいため，容易に実現できる．

4.3　スクリプト言語の API

ネットワークの監視，およびゲスト OS の監視をスクリプト言語で行う上で，スクリプト言語の API を検討する．ネットワーク監視用 API，OS 監視用 API を (**表 1**, **表 2**) に示す．

これらの API 名はオブジェクトであり，hooks メンバをもつ．hooks はリスト型で，イベントが発生した時に hooks 内の対応する処理を呼び出す．hook が呼び出される時に引数が渡され，登録するフック内で処理を記述するのに用いる．フックの登録には hooks の API を呼び出す．(**表 5**)

実際にフック処理を登録するときは

表 1　ネットワーク監視用 API

機能	API 名	引数	フックの引数
L2 パケットの送受信	net.eth	source, destination	Ethernet packet オブジェクト
L2 パケットの受信	net.eth.recv	〃	〃
L2 パケットの送信	net.eth.send	〃	〃
L3 パケットの送受信	net.ip	〃	IP packet オブジェクト
L3 パケットの受信	net.ip.recv	〃	〃
L3 パケットの送信	net.ip.send	〃	〃
TCP パケットの送受信	net.ip.tcp	〃	TCP packet オブジェクト
TCP パケットの受信	net.ip.tcp.send	〃	〃
TCP パケットの送信	net.ip.tcp.recv	〃	〃
UDP パケットの送受信	net.ip.udp	〃,	UDP packet オブジェクト
UDP パケットの受信	net.ip.udp.send	〃	〃
UDP パケットの送信	net.ip.udp.recv	〃	〃

表 2　OS 監視用 API

機能	API 名	引数	フックの引数
物理メモリの読み出し	pmem.read	メモリアドレス	読み込むサイズ
物理メモリへの書き込み	pmem.write	〃	書き込むサイズ, データ
CPU レジスタの読み出し	cpu.read	レジスタ	CPU オブジェクト
CPU レジスタへの書き込み	cpu.write	〃	〃
仮想メモリの読み出し	vmem.read	メモリアドレス	読み込むサイズ
仮想メモリへの書き込み	vmem.write	〃	書き込むサイズ, データ
システムコールの呼び出し	syscall	システムコール	システムコールの引数

表 3　hooks の API

機能	API 名	引数
フックの登録	hooks.add	name, hook
フックの削除	hooks.delete	name, hook
フックの取得	hooks.get	name

```
1 hook = function (...)
2     ...
3 end
4 ...hooks.add("hook1", hook)
```

のようにして追加する．hook には各 API に対応した引数を受け取る関数を実装することで対象となるデータにアクセスできるようにする．hook に渡す引数を**表 4** に示す．

表 4　hook の引数

API 名	hook が受け取る引数
net.eth*	ethernet packet
net.ip*	ip packet
net.ip.tcp*	tcp packet
net.ip.udp*	udp packet
pmem.read	size
pmem.write	size, data

また，hook 内で返す値によってそのイベントがそのまま実行されるかブロックされるか選ぶことができる．

表 5　hook の返り値

API	返り値	意味
net.*	ACCEPT	パケットを通過
	DROP	パケットを廃棄
cpu.*, pmem.*, vmem.*	ACCEPT	アクセスを許可
	DROP	アクセスを拒否
syscall	ACCEPT	実行を許可
	DROP	実行を拒否

5. API の利用例

設計した API を用いて行うことのできる処理の利用例を示す．

(1) TCP 通信の監視

TCP 通信を監視し，あるデータが含まれていた際にそのパケットを DROP する処理を示す．

```
1 hook = function(tcp)
2   if string.find(tcp.data, "malicious[0-9]")
        ~= nil then
3     return DROP
4   end
5
6   return ACCEPT
7 end
8 net.ip.tcp.send.hooks.add("tcpdrop", hook)
```

(2) MAC アドレスの書き換え

全ての TCP の通信を MAC アドレス 01:23:45:67:89:AB 経由にする処理を示す．

```
hook_out = function(eth)
  -- allow if not tcp
  if eth.ip.tcp == nil then
    return ACCEPT
  end

  -- rewrite dst mac address
  eth.dst = "01:23:45:67:89:AB"
  return ACCEPT
end
net.eth.send.add("proxy", hook)
```

これによって TCP 通信が必ず MAC アドレス 01:23:45:67:89:AB (監視用サーバを想定) を経由するプロキシを実現することができる．

(3) プロセス監視との連携

ネットワーク監視用の API と OS 監視用の API を組み合わせることで，不審なプロセスが起動した際に通信を遮断することも可能である．/tmp/以下にあるファイルが起動されたときにネットワークを遮断する処理を示す．

```
denyall = function(eth)
  return DROP
end

execve_hook = function(args)
  filename = args[0]
  argv = args[1]
  envp = args[2]

  filename_s = pmem.read(filename, 0x100)
  if string.match(filename_s, "^/tmp/") ~=
      nil then
    net.eth.hooks.add("denyall", denyall)
  end
end
syscall.hooks.add("netproc", execve_hook)
```

これは/tmp/で始まるバイナリが exeve システムコールで起動されたときにネットワークの通信をすべて DROP し，通信を遮断する．

6. 実装

実装には type1 hypervisor である BitVisor を用いた．これは目標のひとつとしている仮想的なネットワークを構築する上で，各々の物理マシンに対してこのシステムを適用できる方がより有用なためである．また BitVisor は VMM 本体のサイズが非常に小さく，KVM や Xen などの高機能な VMM に比べてバグが発生しにくいため，セキュリティ面でも優れている．BitVisor は単一の ELF ファイルを生成し，これを Bootstrap ローダーとして読み込ませることで VMM を起動するが，これはネットワーク経由でも可能である．すなわち，配布サーバーを設置してネットワーク上のクライアント PC を VMM で起動することができる．

移植する言語においては Python,Ruby,Lua,Lisp を，処理系の大きさ・移植の容易性から評価し，試験的に Lua を選択した．また Lua は組み込み機器における実績が多くあり，オブジェクト指向で今回の API 設計に適している．

6.1 BitVisor 上の実装

BitVisor は VMM の他に，process と呼ばれる名前空間が分離された状態で処理を実行するための機構が実装されており，BitVisor に備わっているディスク暗号化，VPN などの機能はこの process を利用している．VMM と process は分離されているため，通常は process から VMM 側の処理を実行することはできない．これを呼び出せるようにするために，msg という仕組みを通じて process と VMM 間でプロセス間通信のような経路を以下のような手順で確保する．(図 **6.1**)

(1) VMM に process から呼び出す処理を登録する
(2) msg に myfunc を登録する
(3) process が処理を呼ぶために msg を開く
(4) 開いた msg を通じて VMM の処理を呼び出す

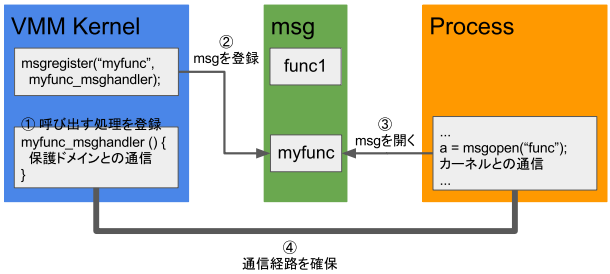

図 3 BitVisor における VMM と process 間の通信の様子

この機構により，process がクラッシュしたとしても VMM 本体には影響を及ぼさず，致命的なクラッシュを防ぐことができる．LVisor を実装する上では，言語処理系の本体は process 側に組み込み，処理系が落ちても本体を巻き込まないようにする．

6.2 BitVisor のビルドシステム

BitVisor 上に別のライブラリを組み込むにあたって，複数の process から同じライブラリを使用できる必要が生じる．例えば，libc などの汎用的なライブラリはもちろん，新たに実装した監視用の API は全ての空間から呼び出せ

ることが望ましい．

BitVisor の VPN やストレージ暗号化機能を見てみると，本体の処理はそれぞれ vpn, storage 以下にあり，process/vpn.c や process/storage.c 内では本体側の処理に関数をリンクしていることがわかる．これはビルド時に解決されて実行コードが直接呼び出せるようになるため，process の名前空間で本体側に実装されているコードを呼び出すことを可能にしている．LVisor においてもこの方法を採用する．Lua や LibVMI は VMM 本体の領域に移植し，process からこれらのコードを利用する．[5] では process 空間に Lua および LibVMI を移植していたため，VMM 本体からの呼び出しが困難だったが，この構成に変更することで解決される．また，libc を VMM 本体側に組み込むことでより多くのライブラリを追加できるようになる．

6.3 言語処理系の移植と BitVisor の変更

API の実装にあたっては Lua 本体に直接実装し，BitVisor のビルド時に直接リンクすることで処理系を組み込む．しかし process 環境下で動作する処理は一切の外部ライブラリを持たないため，最低限必要な関数 (**表 6**) はあらかじめ BitVisor 側で用意されている．

表 6 process 向けに用意されている関数

機能	代替関数名
標準出力	printf, putchar 等
標準入力	lineinput
メモリ確保	alloc
メモリ再確保	realloc
メモリ解放	free

また Lua は libc を用いるため，同様に libc も組み込む必要がある．LVisor では musl-libc を用いている．Lua は libc の他に libm も必要とするが，musl は同様に libm を内包するため適している．

これらの編集に伴って，BitVisor 全体で約 46000 行中 1993 行を修正し，Lua に新規に約 700 行追加した．

7. 評価

LVisor の各種機能におけるオーバーヘッドを計測し，性能評価を行う．今回はネットワーク機能の実装が未完のため，LVisor に組み込んである Lua の性能評価を行う．LVisor においてベンチマーク計測用の process を作成し，dbgsh から起動することで各ベンチマークを計測する．

7.1 評価環境

評価を行う環境を**表 7** に示す．

表 7 評価環境

CPU	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
RAM	7.7GB
OS	Debian8 Jessie

7.2 評価項目および結果

7.2.1 Lua の実行

VMM に組み込んだ Lua の実行によるオーバーヘッドを計測するために，”hello”を出力するだけの処理を，C 言語と Lua で記述したものそれぞれについてその実行時間を比較する．それぞれを 100 回実行し，その平均実行時間を計測する．計測には VMM Kernel の cpu_get_time を mvmon driver 経由で呼び出し，ホストマシンの CPU クロック数を基に算出された時間を用いる．実行結果を**表 8** に示す．

表 8 標準出力の実行時間

言語	平均実行時間 (ms)
C	0.321
Lua	0.333

7.2.2 ループ処理

ループ処理についても同様に計測を行う．計測対象とするコードを以下に示す．

```
1 int sum = 0;
2 for(i = 0; i <= 0x100000; i++) {
3   sum += i;
4 }
```

```
1 sum = 0
2 for i = 0, 0x100000 do
3   sum = sum+i
4 end
```

計測結果を**表 9** に示す．

表 9 for ループの実行時間

言語	平均実行時間 (ms)
C	0.657
Lua	78.708

7.2.3 メモリアクセス

LVisor からゲストマシンの物理メモリを取得するときのオーバーヘッドを計測する．移植した Lua の API (memread, memwrite) を用いて 1 ページ分の読み書きを行い，その実行時間を計測した．結果を**表 10** に示す．

表 10　メモリアクセスのベンチマーク

アクセス	実行時間 (ms/1 ページ)
Read	0.397
Write	0.387

7.2.4　Lua を用いたパターンマッチング

Lua 自身の機能を用いてパターンマッチングを行う際の性能を評価するために，`string.find` を用いて特定のバイト列の探索を行う．物理メモリ空間から ELF のヘッダ (`"\x7fELF"`) を検索する．検索対象アドレスは 0x0-0x100000 までで，検索開始からヒットするまでの時間を計測する．結果を**表 11** に示す．

表 11　ELF ヘッダのマッチング

実行コード	実行時間 (ms)
`string.find(a, "\x7fELF")`	0.787

8. おわりに

本論文ではネットワークの通信を制御するための仕組みを考案し，LVisor に実装するための設計を行った．現状では実装が未完成のため定量的な評価を行うことができなかったが，BitVisor は準パススルー型の VMM であるため高速に動作することが期待できる．パケットへの介入を言語処理系で行えるため，API の利用例に挙げた例の他に，仮想 NAT の実現や VPN による通信路の保護などが可能である．

現在，ネットワーク制御によるオーバーヘッドを計測するために実装を進めている．

参考文献

[1] IPA 独立行政法人 情報処理推進機構, ”未知ウイルス検出技術に関する調査”, 2004
[2] ”Lua スクリプト機能”, `https://network.yamaha.com/setting/router_firewall/monitor/lua_script`
[3] `http://www.rekall-forensic.com/`
[4] `http://libvmi.com/`
[5] 市川遼, 並木美太郎: ”言語処理系を組み込んだ VMM による OS の効率的な監視システム”, 第 140 回 システムソフトウェアとオペレーティング・システム研究会, 2017
[6] 岩村誠, 伊藤光恭, 村岡洋一: ”機械語命令列の類似性に基づく自動マルウェア分類システム”, 情報処理学会論文誌, vol. 51, pp. 1622-1632, 2010
[7] 岩本一樹, 和崎克己: ”静的解析により抽出された API 推移に基づくマルウェアの分類”, 情報処理学会論文誌, vol. 54, pp. 1199-1210, 2013
[8] S. Cesare, Y. Xiang: ”Classification of malware using structured control flow”, Eighth Australasian Symposium on Parallel and Distributed Computing, vol. 107, pp. 6170, 2010
[9] 田辺瑠偉, 笠間貴弘, 吉岡克成, 松本勉: ”重要情報へのファイルアクセス失敗挙動に基づく情報探索型マルウェア検知手法”, 情報処理学会論文誌, vol. 57, pp. 597-610, 2016
[10] 川古谷裕平, 岩村誠, 針生剛男: ”テイント伝搬に基づく解析対象コードの追跡方法”, 情報処理学会論文誌, vol 54, pp. 2079-2089, 2013
[11] 大月勇人, 瀧本栄二, 齋藤彰一, 毛利公一: ”マルウェア観測のための仮想計算機モニタを用いたシステムコールトレース手法”, 情報処理学会論文誌, vol. 57, pp. 2034-2046, 2014
[12] Henderson, Andrew and Prakash, Aravind and Yan, Lok Kwong and Hu, Xunchao and Wang, Xujiewen and Zhou, Rundong and Yin, Heng: ”Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform”, Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 248-258, ACM, 2014.
[13] Lengyel, Tamas K and Maresca, Steve and Payne, Bryan D and Webster, George D and Vogl, Sebastian and Kiayias, Aggelos: ”Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system”, Proceedings of the 30th Annual Computer Security Applications Conference, pp. 386-395, ACM, 2014.
[14] 中田健介, 青木一史, 神谷和憲, 角田進, 大嶋嘉人: ”動的解析結果に基づく共通的なマルウェア通信パターン抽出技術の検討”, コンピュータセキュリティシンポジウム 2015 論文集, vol. 2015, pp. 318-325, 2015
[15] 田中功一, 堀川博史, 峰野博史, 西垣正勝: ”ログ解析によるマルウェア侵入検知手法の提案”, マルチメディア、分散協調とモバイルシンポジウム 2014 論文集, vol. 2014, pp. 522-529, 2014
[16] 淵上智史, 長谷川皓一, 山口由紀子, 嶋田創, 高倉弘喜: ”マルウェア感染拡大抑止に向けたネットワーク型動的解析システム”, 2016