Vol.2017-OS-140 No.18
2017/5/17

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

市川 遼[1,a)]　並木 美太郎[1]

概要：コンピュータウイルスやマルウェアなどの不正ソフトウェアを検知する方法として，プログラムに固有のシグネチャを予め用意してマッチングに利用する方法が挙げられるが，検知できるパターンに限界があり，また動的にコードを展開するようなマルウェアに対しては無力である．
本研究ではVMMに言語処理系を組み込み，カーネルより高い権限で動作可能な監視システムLVisorを提案する．このシステムではスクリプト言語を用いてそれを監視ルールとすることで，従来のパターンマッチングでは実現できなかった，データに対して任意の処理を行うことを可能にしている．またVMMにはBitVisorを用いているため，実機に近いパフォーマンスで動作する．LVisorを試作し言語処理系による実行時間を計測したところ，ネイティブコードと比べて遜色ない程度であった．今後ネットワーク通信を監視，干渉できるようにするのが課題である．

1. はじめに

サイバー攻撃に用いられるマルウェアに対しては，シグネチャマッチと呼ばれる，データに含まれている特徴的なデータ列 (シグネチャ) を用いて判定を行う方法が最も一般的であり，広く使われてきた．しかし近年のマルウェアの対解析機能は進化してきており，コードを圧縮して実行時に展開することでシグネチャマッチによる検知を迂回するものや，実行パスを複雑にすることで処理の流れをわかりにくくするものが登場してきた．このような検体に対してはシグネチャマッチおよび静的解析は無力であるため，サンドボックス環境などを用意して動的解析を行うのが最も有効な手段だが，実行環境を確認して解析されていることを検知するようなマルウェアも存在するため確実な方法ではない．またシグネチャマッチでは異なる検体毎にその特徴を保持する必要があるため，データ量が膨大になってしまうという欠点がある．

そこで本論文では，VMMに言語処理系を組み込み，マッチングのルールをスクリプト言語で記述する手法を提案する．本提案により，類似したシグネチャをまとめて記述したり，従来の方法では実現できなかったデータに対する構造的な解析を行うことを目的とする．また，VMM上にLibVMI[3]を移植することによりセマンティックギャップを解決し，ゲストOS内の情報にVMM側からアクセスすることを可能にする．

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

2. 従来手法と課題

2.1 VMM

VMMを用いたサンドボックスによってOSの挙動を監視する研究には，QEMUやXenなどが用いられる．QEMUではDECAF[6]，XenではLibVMI[3]を用いたDRAKVUF[7]，ネットワークのデータに焦点を当てたものなどが挙げられる[10]．

DECAFではテイントタグ伝搬による解析手法を用いており，これをQEMUのプラグインとして実装している．Windows APIをフックしてその返り値を書き換えたり，スタックの操作を行うことなどが可能であるが，テイントタグをDBI中に生成されるコードに付与しているため，Intel VTなどの仮想化支援機構を利用できない．このためエミュレーションをすべてCPUで行う必要があり，処理量が膨大になるため動作が緩慢である．

DRAKVUFではLibVMIを用いてsemantic gapを解決し，ゲストOS内の仮想アドレスと物理アドレスを相互変換することを可能にしている．これと，Dom0側から提供されるゲストOSのカーネル上の構造体やオフセットの情報が記されたJSONファイルを用いてプロセス監視，ファイル監視などを機能を実現している．しかしXenを動作させるためには専用のカーネルが必要であり，また物理マシンの多くのリソースが占有される．

また，仮想化技術は近年クラウド環境における利用が増えてきており，クラウド環境におけるセキュリティの向上に関する研究も活発に行われている [9][11]．特にゲスト OS 内にエージェントを設置しない場合，OS 上でどの API を呼び出したかという情報はメモリや CPU の情報だけでは得ることができない．これを semantic gap といい，VMM を用いて OS 内部の挙動を観測する際に解決しなければならない問題の一つとして知られている．

本研究においては，DRAKVUF で用いられている LibVMI の機構を用いる．LibVMI はもともと Xen に備わっていた XenAccess[4] という VMI の後継であり，近年様々な研究でも用いられており，前述したクラウド環境での仮想環境における VMI の手段として用いている研究も数多くある [13][14][15].

2.2　ルールの記述

マルウェアに限らず脅威を検知するシステムには，どのような特徴を危険とみなすかという情報が必要である．例えば，マルウェアが実行されるときはマルウェア本体がメモリ上に展開され，場合によっては外部のサーバーに通信を行うため，ネットワーク通信やメモリ上のデータなどが手がかりになる [12]．一般的なマルウェアの検知においては，そのデータ内に出現する特徴的なバイト列 (シグネチャ) を登録しておき，検索対象となるファイル群にシグネチャが含まれているものをマルウェアとみなす．このとき，あるシグネチャがマルウェアでないものに含まれている可能性はゼロではないため，AND/OR などの論理演算を用いて条件を記述することで偽陽性と判断されるのを防ぐ．

このルールを表現するために各検知エンジンは DSL (Domain Specific Language) を定義してそのパーサーを実装しているものがほとんどである．マルウェア研究者向けの検体分類ツールとして用いられている Yara[8] も独自フォーマットが定められており，定数の定義と定数を用いた AND/OR 条件記述が可能である．

しかしこれらはパターンマッチングのために設計されたものであり，その記述力には限界がある．また，定義したデータ以外は一切用いることができないため，拡張性を著しく欠いている．

3.　目標

そこで本論文では，従来の VMM におけるリソース占有問題，表現力・拡張性の低い検知ルールの表現法を改善することを目標とする．

従来の VMM では性能不足なマシンを用いたときにゲスト OS 内の処理能力が著しく低くなってしまうため，なるべくリソースを必要としない VMM を検討する．また，検知エンジン向けに設計された DSL ではその表現力に限界があるため，特定の処理ではなく任意の処理が記述可能である，プログラミング言語を検知ルールとして用いることで，より柔軟な監視が行えるシステムを実現する．

4.　提案手法

4.1　システムの構成

本システムの全体構成を図 **4.1** に示す．

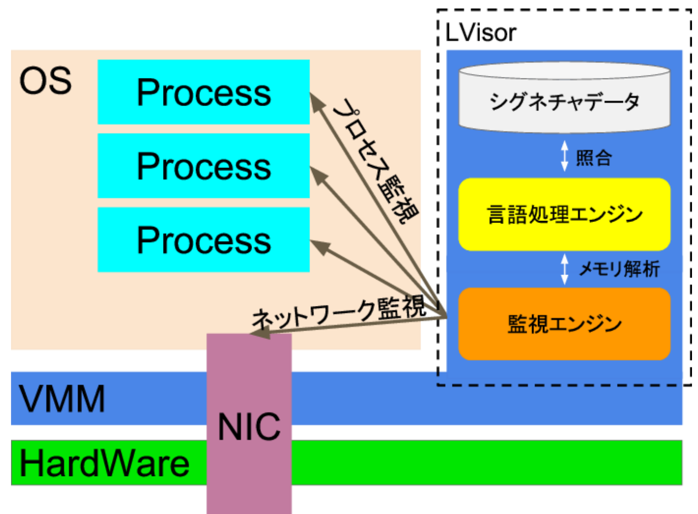

図 1　LVisor の全体構成

VMM 上に実装された監視エンジンがゲスト OS から情報を取得し, 解析対象とするデータを抽出して言語処理系に渡す. 言語処理系は受け取ったデータをあらかじめ用意したシグネチャデータを用いて, 悪性コードが含まれていないか検査する.

それぞれのコンポーネントの役割は表 **1** に示す通りである．

表 1　LVisor の各コンポーネントの役割

名前	機能
監視エンジン	ゲスト OS のデータ取得
言語処理エンジン	シグネチャデータの実行
シグネチャデータ	監視ルール (実行可能スクリプト)

4.1.1　監視エンジン

監視エンジンは VMM を通じてゲスト OS のメモリ，NIC 上のデータを取得する．メモリ全体を取得すると非常に多くの時間がかかるため，ゲスト OS のプロセスが使用している領域だけを対象とする．取得したデータは言語処理系エンジンが後述するシグネチャと合わせて解析を行う．

監視エンジンの構成を図 **2** に示す．

監視エンジンは lvmon core と CPU,memory,NIC にアクセスするためのドライバー群 (lvmon driver)，監視するイベントを管理するための hook monitor からなる．

ⓒ 2017 Information Processing Society of Japan

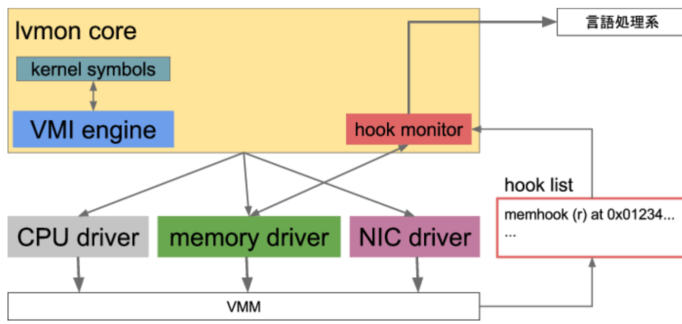

図 2　監視エンジンの構成

lvmon driver は VMM からゲスト OS の CPU，メモリ，NIC などにアクセスするためのインターフェースであり，LVisor からの全てのアクセスはここを通じて行われるため，VMM に合わせて最適に動くように調整されている必要がある．

hook monitor は特定のイベントが発生したときに VMM から lvmon に処理を移すための機構である．特定領域へのメモリアクセスなどの監視対象イベントが発生すると，VMM は登録されたフック関数を実行し，どのようなアクセスが発生したかという情報を hook list に書き込む．hook monitor は hook list に書き込まれたイベント情報を見て，言語処理系エンジンを呼び出して処理を行う．

lvmon core は監視エンジンの核となる部分であり，以下に示す機能を提供する．

- メモリアクセス監視: ゲスト OS の物理メモリ上の特定アドレスへのアクセスを監視する
- システムコール監視: ゲスト OS で発行されるシステムコールを監視する
- プロセス監視: ゲスト OS のプロセスリストを監視する
- 動的メモリ監視: プロセスが動的に生成したメモリの内容を監視する
- NIC 監視: ゲスト OS の NIC を流れるデータを監視する

特に動的メモリ監視は近年のマルウェアが用いるプロテクタと呼ばれる技術による実行コード隠蔽を破るのに有用である．マルウェア作成者はマルウェア開発のコストを減らすため，過去に作成されたマルウェアのコードを一部流用することがあるが，そのままではほとんどのアンチウイルスソフトに検知されてしまう．そのため，プロテクタを用いてコードを暗号化し，実行時に展開することで実行ファイルに対するパターンマッチングによる検知を迂回する．図 3 展開された後のメモリの内容に対してパターンマッチングを行うことで，単にパッカーが適用されているだけの無害なプログラムに対する誤検知を未然に防ぐことができる．

4.1.2　言語処理系エンジン

言語処理系エンジンは VMM のドライバを用いてメモリ，ネットワーク通信，OS レベルの API 呼び出しを監視する．従来は得られたデータに対して簡単なシグネチャマッチを行うことしかできなかったが，言語処理系を用いることで任意の処理をデータに対して適用することが可能である．例えば，bss 領域に注目すべきデータが入っているが実行領域が巨大で全てを探索するのが非効率的である場合に，実行ファイルのヘッダをパースして bss 領域だけ取り出してシグネチャマッチを行う，などといった効率化も可能である．また，ネットワーク上のデータに対しては，SSH や RDP など特定のプロトコルを用いて接続を行うような攻撃が発生したときに，プロトコルを解釈する処理をあらかじめ実装しておくことで通信内容を調査することができる．

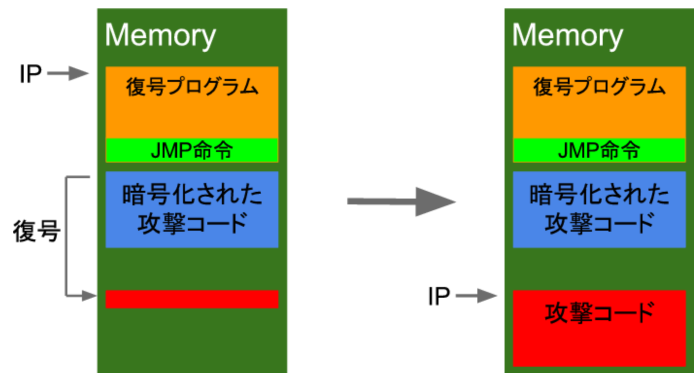

図 3　攻撃コードを動的に復号する様子

これらを考慮すると，ルールは動的に追加可能であることが望ましいため，組み込む言語処理系エンジンは動的にコードを評価することができるインタプリタ型である．また，VMM はベアメタル上で動作するため，なるべく外部ライブラリに依存しないサイズの小さい処理系が望ましい．これらの項目について，micropython, mruby, Lua, Embeddable Common-Lisp を比較する．これらはそれぞれ Python, Ruby, Lua, Lisp の処理系を持っており，組み込み向けとしての実績がある．

まずサイズの比較を行う．それぞれのインタプリタをビルドし，動的にリンクしたサイズを調べる．これは外部ライブラリを除いた，言語処理系の純粋なサイズを比較するためである．表 2 に各インタプリタのサイズを示す．

表 2　インタプリタのサイズ

言語処理系名	バージョン	サイズ
micropython	v1.8.7-77-gbf51200	405KB
mruby	1.2.0	1.5MB
Lua	5.3.3	236KB
Embeddable Common-Lisp	16.1.3	26KB

Embeddable Common-Lisp (ecl) が最も小さく，Lua がそれに続く形となった．続いて，ldd コマンドを用いて各インタプリタが依存するライブラリを表 3 に示す．ここ

ⓒ 2017 Information Processing Society of Japan

で，ld-linux および linux-vdso はインタプリタの機能に関わらないライブラリであるため，対象外とする．

表 3 依存ライブラリ

言語処理系名	ライブラリ
micropython	libc, libm, libdl, libpthrad, libffi
mruby	libc, libm, libreadline, libncurses, libtinfo
Lua	libc, libm
ecl	libc, libm, libdl, libpthread, **libecl**

ecl が使用する libecl というライブラリは ecl の処理系を担っているライブラリであり，そのサイズは約 11MB であった．また，libffi,libgmp がリンクされているため ecl は計 6 つの外部ライブラリに依存していることになる．

以上を踏まえて，使用する言語の検討を行う．まずサイズであるが，ecl は libecl に依存しているため実際は最もサイズが大きい．ecl を除外すると Lua が約 236KB, micropython が 405KB と続き，mruby が 1.5MB と少し大きい．次に依存するライブラリに注目すると，micropython および ecl は libpthread を使用しているが，これはカーネルによって `sys_prctl` および `sys_arch_prctl` がサポートされている必要がある．VMM によっては実装されていないものもあるため，このライブラリを移植するのは困難である．mruby が依存している libreadline, libtinfo, libncurses は，ターミナル I/O 周りのライブラリであるため移植は可能であるが，Lua と比較するとインタプリタのサイズ，依存ライブラリの少なさ共に劣っている．よって本システムに組み込む言語処理系として Lua を選択する．LVisor では Lua に対して以下の機能を持つ API を提供する．

- 物理メモリの読み書き
- CPU レジスタの読み書き
- 現在のプロセスの取得
- プロセス一覧の取得
- メモリアクセスのフック
- システムコールのフック
- ネットワーク通信のフック (通信先を指定)

これらを用いて言語処理系 (Lua) が適切な処理を行う．シグネチャデータが言語処理系によって処理される様子を**図 4** に示す．

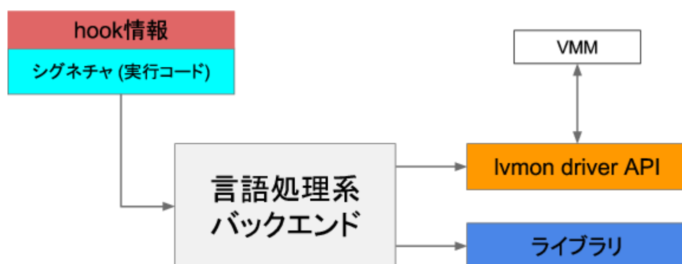

図 4 言語処理系とシグネチャデータ

4.1.3 シグネチャデータ

シグネチャデータはマルウェアを検知するためのルールに相当するものであり，その実態は Lua で記述されたプログラムである．Lua の関数と監視するイベントを紐付けて登録しておき，監視対象のイベントが発生すると登録した関数が呼ばれるという，イベント駆動型のロジックで動作する．**表 7** に示した API を用いて処理内容を記述し，ファイルに対する単なるシグネチャマッチだけでなく，ゲスト OS 内のプロセス情報などを使用することができるため，より柔軟なルールを記述することが可能である．シグネチャの例を program 1 に示す．

program 1

```
1  handler = function(args)
2    proc = lvmon.getproc()
3    memdata = lvmon.memread(proc.memory,
                                 proc.mem_size)
4    idx = string.find(memdata,
5                          "suspicious[0-9]")
6    if idx then
7      lvmon.memwrite(proc.memory,
8      "\x00"*proc.mem_size) -- crash process
9    end
10 end
11 lvmon.hook.syscall("sys_execve", handler)
12
```

この例では `sys_execve` に対してフックを設定する．システムコール `execve` が実行されたときに現在のメモリを取得し，”suspicious”+(0-9 の数字) という文字列が含まれていた場合はプロセスメモリを NULL バイトで埋めてクラッシュさせるというものである．

4.2 実装

使用する VMM には，Thin Hypervisor として開発されている BitVisor を用いる．本システムで使用したバージョンは 1.4 である．

BitVisor では CPU の仮想化支援機構及び x86_64 の命令を用いて VMM を実装している．Intel VT-x および AMD SVM に対応しており，現在流通している多くのマシンで動作する．

4.2.1 BitVisor の保護ドメイン

BitVisor は VMM Kernel, 保護ドメイン, ゲスト OS (VM) に層が分かれている．VMM Kernel が最も強い権限を持ち，保護ドメインはその下に属する形となっている．ゲスト VM は VMM Kernel が管理しており，保護ドメインとゲスト VM は互いに直接干渉することができない．保護ドメインはプロセスに似た特徴を持っており，複数生成

ⓒ 2017 Information Processing Society of Japan

が可能かつメモリ空間が互いに干渉しない．また保護ドメインから別の保護ドメインを起動することが可能である．保護ドメイン間および VMM Kernel-保護ドメイン間で通信を行う手段はメッセージング I/F に限定されており，呼び出し側はメッセージング用の経路に対してハンドラを設定する．これによって定義したもの意外の処理を実行できず，VMM Kernel がセキュアに保たれる．BitVisor における各コンポーネント間の通信の様子を図 **5** に示す．

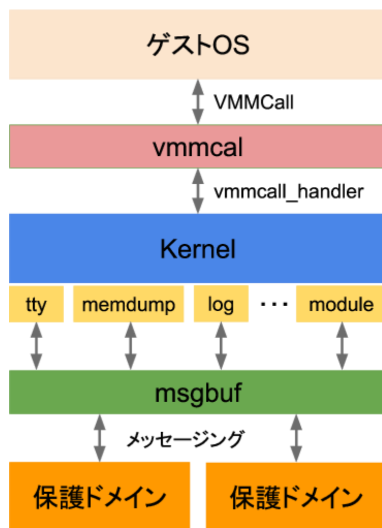

図 **5**　VMM Kernel, 保護ドメイン, ゲスト OS 間の通信

BitVisor の保護ドメインは process と呼ばれ，プラグインなど別の機能を実装するのに用いる．process 間で通信を行うには msg と呼ばれるメッセージング I/F を用いてデータの送受信を行う．VMM Kernel 部分に処理を追加した場合，バグがあったときに VMM 全体がクラッシュする危険性があり致命的であるため，VMM Kernel 側の情報を必要としない機能については process で実装を行い，VMM Kernel 側で必要な情報のみを msg を通じて取得する．

4.2.2　各 driver の実装

msg を用いて lvmon driver の実装を行った．提供する API を表 **4** に示す．

表 **4**　lvmon driver の仕様

API 名	機能
GPMEM_READ	ゲスト OS の物理メモリのデータを読み込む
GPMEM_WRITE	ゲスト OS の物理メモリにデータを書き込む
GPMEM_REG_READ	ゲスト OS の CPU レジスタの値を取得する
GPMEM_REG_WRITE	ゲスト OS の CPU レジスタに値を書き込む
GPMEM_GET_MEMSIZE	ゲスト OS の物理メモリのサイズを取得する
GPMEM_GET_CPU_TIME	VMM Kernel の CPU 時間を取得する

4.2.3　Lua の移植

Lua を BitVisor へ組み込むにあたり，幾つかの技術的課題が発生した．ここではその解決方法について述べる．

Lua の依存ライブラリ

表 3 で示したように，Lua のビルドには libc 及び libm が必要となる．libc は Linux の基本的な機能を，libm は主に数学関数を提供するライブラリである．ベアメタル環境においてはカーネルが存在しないため，必要な機能は開発者が実装しなければならない．BitVisor においても入出力，メモリ管理などの関数は用意されているが，同時にこの環境においては必ず用意された関数を使用しないと正しく動作しないことを意味しており，libc などに依存した別のソフトウェアを移植する際には注意が必要である．ただしシステムコールを用いる関数以外はそのまま移植することができ，libm や文字列操作などがこれに該当する．

BitVisor にはファイルシステムが存在しないため，動的リンクでライブラリを動的に呼び出すのは不可能である．したがって全てのプログラムは静的リンクされている必要があり，外部ライブラリに依存するソフトウェアを動作させるためには，そのライブラリを静的リンクでバイナリ中に埋め込まなければならない．LVisor で用いる libc として，glibc,musl,dietlibc,uClibc について検討を行ったが，musl 以外 Lua に対する静的リンクが行えなかったため musl を用いる．musl は同時に libm も内包しているため，このライブラリのみで Lua のリンクを行うことができた．

Lua で使用される関数の修正

Lua は process として実装を行う．これは，万が一 Lua がメモリ不足などの原因でクラッシュしても VMM Kernel に致命的な影響を与えるのを防ぐため，また BitVisor の保護ドメインに基づいたセキュリティモデルを破壊しないためである．

process は BitVisor 環境下で動作するため，幾つかの関数は提供された API に置き換えなければならない．表 **5** に対象となる関数の種類と process 中で用意されている代替関数を示す．

表 **5**　process

関数の種類	process 内の代替関数
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

メモリ管理については，process を記述するソースコード中にヒープのサイズを

ⓒ 2017 Information Processing Society of Japan

定義し，それを process/lib/lib_mm.c 中の extern heap[HEAPSIZE]，heaplen が吸収する．代替関数 alloc は heap からメモリを切り出しそのポインタを返す．

Lua の移植においては，libc 中の関数について 表 5 に含まれるもの全てを置き換える必要がある．libc 中で修正すべき関数の一覧を**表 6** に示す．

表 6 libc の修正箇所

修正箇所	修正内容
stdio:printf	process の printf に置き換える
stdio:snprintf	process の snprintf に置き換える
stdio:vsnprintf	process の vsnprintf に置き換える
stdlib:malloc	process の alloc に置き換える
stdlib:realloc	process の realloc に置き換える
stdlib:free	process の free に置き換える

置き換えられる関数と置き換える関数が同一シンボルである場合，libc のソースコードの定義箇所の前に extern を加えることで，外部オブジェクトのシンボルを参照するようになるため，このように修正すると libc の挙動を変えることができる．また，Lua のインタプリタ lua.c においては fgets や fputs などの関数を用いているため，適宜修正を行う．

これにより Lua のインタプリタが process として起動するようになった．
また，luaL_dostring(lua_State *L, char *expr) を実行することで文字列 expr を Lua のコードとして評価可能である．また，BitVisor 上で使用できない機能 (system 関数実行，ファイル I/O など) を標準で有効にしているため，一部の機能を Lua のビルド時に無効化した．

4.2.4 LibVMI の移植

LibVMI は LVisor の機能を実現するために不可欠であるため，Lua と同様に BitVisor 上に移植する必要がある．LibVMI は Glib, json-c に依存しているため，これらのライブラリも移植する必要がある．

Glib に関しては LibVMI が必要とする GHashTable, GList, GNode, GQueue, GSList を移植した．また整合性を取るために gmalloc など一部の処理を書き換えて，ビルドが通る状態にした．

json-c はゲスト OS カーネルのシンボル情報をパースするのに必要であるが，対象とする JSON ファイルは rekall を用いて生成したものであり，Linux のもので 3.3MB と非常に大きい．json-c は内部で malloc を用いているためこのような巨大なファイルをパースするとメモリ不足で必ずクラッシュしてしまう．そこで malloc を用いない JSON パーサを独自に作成し，組み込んだ [5]．最低でも rekall が出力する JSON をパースすることができるように実装したため，JSON の仕様全てには対応していないが，その分 JSON パーサのサイズを小さく抑えることができている．

4.2.5 Lua で提供する API

VMM に組み込む Lua から提供する API として，以下のようなものを設計，試作した．

表 7 VMM の Lua が提供する API

機能	API 名
物理メモリの読み出し	memread
物理メモリへの書き込み	memwrite
CPU レジスタの読み出し	regread
CPU レジスタへの書き込み	regwrite
プロセスの取得	getproc
アドレスアクセスのフック	hook.address
システムコールのフック	hook.syscall

VMM に Lua を組み込む際にこれらの機能を実装したものを追加し，Lua のデフォルトライブラリとして bitvisor を使用可能にする．実装した API は bitvisor.memread のようにして呼び出すことができる．

5. 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．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．

5.1 評価環境

評価を行う環境を**表 8** に示す．

表 8 評価環境

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

5.2 評価項目および結果

5.2.1 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 lvmon 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 9** に示す．

表 9　標準出力の実行時間

言語	平均実行時間 (ms)
C	0.321
Lua	0.333

5.2.2　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 program 2, program 3 に示す．

program 2

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

program 3

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

計測結果を**表 10** に示す．

表 10　for ループの実行時間

言語	平均実行時間 (ms)
C	0.657
Lua	78.708

5.2.3　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 11** に示す．

表 11　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

5.2.4　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 12** に示す．

表 12　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

5.3　考察

Lua の処理系は for ループなどのプログラミング言語特有の処理が含まれると大きなオーバヘッドが発生している．これは Lua がループ処理を LuaVM 内で行うことによるものだと考えられるが，スクリプト言語であれば必ず発生するものである．一方で出力だけを行う際に生じる処理は，`print` を出力命令に変換し LuaVM 内で実行するというものであるが，結果より Lua による影響は 12 μ s 程度であり，C 言語とほぼ変わらない速度といえる．

メモリアクセスについては，読み込み，書き込み共に 0.4ms 程度と，非常に高速であった．これは BitVisor の `msg` を用いているためであり，ボトルネックが抑えられていると考えられる．特にメモリアクセスは頻繁に発生すると予想できるため，この値はシステムの性能に直結する．

Lua を用いたパターンマッチングも高速であったが，これは LuaVM による高速化の恩恵と考えられる．Lua は `string` のパターンマッチング処理以外にも多くの機能を備えているため，より柔軟かつ高速なルールを記述することができるのではないかと期待される．

また，これらはいずれも BitVisor の process として動作するため，あらかじめ BitVisor に割り当てた以上のメモリを消費しない．BitVisor は標準で 128MB のメモリを使用するが，今回は Lua を動作させるために 512MB のメモリを割り当てた．これによってゲスト OS が使用できるメモリ量は多少減少するものの，動作に影響は見られなかった．このことから，BitVisor 上でより多くメモリを確保することで，擬似的なファイルシステムが構築可能で，この領域にシグネチャデータを置くことで，ディスク I/O のボトルネックを考慮しなくて良い高速なシステムが実現可能であると考えられる．

6.　おわりに

本研究では，VMM と言語処理系を組み合わせることにより，新しいルールの表現方法を提案した．この手法ではより複雑な条件を記述することができ，VMM と連携することでゲスト OS から検知されにくい監視システムとなっている．また VMM に Thin Hypervisor を用いることで実機に近いパフォーマンスで動作しており，Xen などの多くのリソースを必要とする VMM で生じた問題を解決できたと考えられる．

パターンマッチングによるマルウェア検出の精度は非常に高く，今後も使われ続ける手法である．近年のサイバー攻撃においては，新規のマルウェアに比べて既存のマルウェアを変換して用いるものも多く，静的解析の弱点を動

的解析でうまく補うことが効率の良い解析につながる．本研究においてはパターンマッチングのルール拡張という切り口から，静的解析および動的解析の弱点を互いに補うことができたのではないかと考えられる．

今後，ネットワーク監視部分の実装を行い，通信を遮断する処理をルール中に記述できるようにするのが課題である．この機能が実現されると感染後の 2 次被害を防ぐことが可能になり，より効率的な監視を行えるようになる．

参考文献

[1] QEMU, http://www.qemu-project.org/
[2] The Xen Project, https://www.xenproject.org/
[3] LibVMI, http://libvmi.com/
[4] XenAccess, https://sourceforge.net/projects/xenaccess/
[5] minijson, https://github.com/icchy/minijson
[6] 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.
[7] 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.
[8] YARA - The pattern matching swiss knife for malware researchers, http://virustotal.github.io/yara/
[9] Kirat, Dhilung, Giovanni Vigna, and Christopher Kruegel. "BareCloud: Bare-metal Analysis-based Evasive Malware Detection." USENIX Security. Vol. 2014. 2014.
[10] Johnston, Reece, et al. "Xen Network Flow Analysis for Intrusion Detection." Proceedings of the 11th Annual Cyber and Information Security Research Conference. ACM, 2016.
[11] Win, Thu Yein, Huaglory Tianfield, and Quentin Mair. "Virtualization security combining mandatory access control and virtual machine introspection." Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on. IEEE, 2014.
[12] Santos, Igor, et al. "Opcode sequences as representation of executables for data-mining-based unknown malware detection." Information Sciences 231 (2013): 64-82.
[13] Wang, Zhe, et al. "CloudAuditor: A Cloud Auditing Framework Based on Nested Virtualization." Cyber Security and Cloud Computing (CSCloud), 2016 IEEE 3rd International Conference on. IEEE, 2016.
[14] Kumara, MA Ajay, and C. D. Jaidhar. "Virtual machine introspection based spurious process detection in virtualized cloud computing environment." Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), 2015 International Conference on. IEEE, 2015.
[15] wook Baek, Hyun, Abhinav Srivastava, and Jacobus Van der Merwe. "Cloudvmi: Virtual machine introspection as a cloud service." Cloud Engineering (IC2E), 2014 IEEE International Conference on. IEEE, 2014.

ⓒ 2017 Information Processing Society of Japan