AMD SEVで保護されたVMのeBPFを用いた高速な監視

上杉　貫太[1]　光来　健一[1]

概要：クラウド上の仮想マシン（VM）内の機密情報がクラウドの内部犯によって盗聴されるリスクが問題となっている．そのため，AMD SEV を用いて VM のメモリを暗号化することにより VM 外からの盗聴を防ぐようになっている．SEV で保護された VM のメモリは VM 外で動作する侵入検知システム（IDS）には直接監視することができないため，先行研究では VM 内で安全に動作するエージェントと通信を行うことでメモリデータを取得し，OS データの監視を可能にしている．しかし，ポインタを用いる OS データを監視する場合，ポインタをたどるたびにエージェントとの通信が必要となり，監視性能が低下する．本稿では，AMD SEV で保護された VM に eBPF プログラムを送り込み，OS データを先読みして一括で取得することにより VM の監視を高速化するシステム eBPFmonitor を提案する．eBPFmonitor では，IDS からの要求に応じて eBPF プログラムを実行することにより，ポインタを用いる OS データを先読みする．IDS からの 1 回の要求でメモリデータを一括で返送することにより，通信のオーバヘッドを削減する．eBPFmonitor を BPF CO-RE と LLView を用いて実装し，監視対象 VM の OS データを取得する性能を調べた．

1. はじめに

クラウド上の仮想マシン（VM）を利用するユーザが増えるにつれ，VM 内の機密情報がクラウドの内部犯によって盗聴されるリスクが問題となっている．そのため，AMD 製の CPU は VM のメモリを暗号化することで VM 外からの盗聴を防ぐ SEV [1] と呼ばれる機能を提供している．SEV の暗号鍵は CPU によって管理されるため，クラウドの内部犯であってもメモリデータを復号できず機密情報の漏洩を防ぐことができる．一方，SEV で VM のメモリを暗号化していても，VM 内でのメモリアクセスに対しては復号が透過的に行われメモリを自由に読み書きすることができる．このように，SEV は VM 内の侵入者に対しては無力であるため，侵入検知システム（IDS）と併用する必要がある．しかし，VM が SEV で保護されていると VM の外にオフロードした IDS は VM のメモリ上の OS データを監視できなくなる．

そこで，VM の外にオフロードした IDS が VM 内で安全に動作するエージェントと通信することでメモリデータを取得し，OS データの監視を行う SEVmonitor [2] が提案されている．SEVmonitor を用いることにより，SEV で暗号化された VM の侵入検知を VM 外部から行うことができる．しかし，監視に用いるメモリデータは IDS が必要とした時に取得されるため，プロセスリストなどのようにポインタを用いる OS データの場合はポインタをたどるたびにエージェントとの通信が必要となり，監視性能が低下する．

本稿では，SEV で保護された VM に eBPF プログラムを送り込み，OS データを先読みして一括で取得するシステム eBPFmonitor を提案する．eBPF は OS にプログラムをロードして性能等を監視するために用いられている Linux の機構である．eBPFmonitor は SEVmonitor と同様に，監視対象 VM 内でエージェントを安全に動作させる．eBPFmonitor のエージェントは IDS からの要求に応じて送り込まれた eBPF プログラムを実行することにより，ポインタを用いる OS データを先読みする．IDS からの一回の要求で OS データを一括で返送することにより，通信のオーバヘッドを減らし VM の高速な監視を実現する．先読みする OS データは IDS ごとに異なるため，eBPFmonitor は IDS ごとに作成した eBPF プログラムを監視対象 VM 内の OS に動的にロードし，ロード時に検証を行うことで安全に実行する．

eBPFmonitor を BPF CO-RE [3] と LLView [4] を用いて実装した．VM 内のエージェントが IDS から OS データの一括取得要求を受信すると，要求に対応するイベントを発生させて eBPF プログラムを呼び出す．eBPF プログラムは終了しない可能性のあるループを実行できないため，有限ループを用いて OS データのポインタをたどり，アドレス一覧を収集する．その後，エージェントは収集した

[1] 九州工業大学
Kyushu Institute of Technology

© 1959 Information Processing Society of Japan

アドレスに対応するメモリデータを OS から取得し，順次 IDS に返送する．eBPFmonitor を用いて実験を行い，監視対象 VM のプロセスおよびカーネルモジュールのリストを一括で取得できることを確認した．また，eBPFmonitor は SEVmonitor よりプロセスリストを 3.1 倍，カーネルモジュールのリストを 4.2 倍高速に取得できることが確認できた．

以下，2 章では SEV で保護された VM を監視する際の問題点と SEVmonitor の監視性能が低下する原因について述べる．3 章では SEV で保護された VM に eBPF プログラムを送り込み，OS データを先読みして一括で取得することにより VM の監視を高速化するシステム eBPFmonitor を提案する．4 章では eBPFmonitor の実装について説明する．5 章では eBPFmonitor の動作確認と性能測定の実験について述べる．6 章で関連研究について述べ，7 章で本稿をまとめる．

2. SEV で保護された VM の監視

近年，組織の内部の管理者などによって機密情報が盗聴される内部不正が問題となっている．それに伴い，クラウド上の VM に保存されている機密情報がクラウドの内部犯によって盗聴されるリスクも高まっている．そのため，AMD 製の CPU は VM のメモリを暗号化することで VM 外からの盗聴を防ぐ SEV と呼ばれる機能を提供している．SEV は VM ごとに固有の暗号鍵を用いて VM のメモリを透過的に暗号化する．暗号鍵の生成と管理は AMD セキュアプロセッサによって行われるため，クラウドの管理者であっても暗号鍵を取得することができず，VM 内の機密情報の盗聴を防ぐことができる．

一方，SEV によって VM のメモリを暗号化していても，VM 内でのメモリアクセスに対しては復号が透過的に行われメモリを自由に読み書きすることができる．そのため，SEV による保護は VM 内の侵入者に対しては無力である．攻撃者は OS などのソフトウェアの脆弱性やマルウェアを利用することで，容易に機密情報を取得することができる．そこで，SEV による VM の保護と侵入検知システム（IDS）による VM 内のシステムの監視を併用する必要がある．IDS を用いることにより，例えば，攻撃者のプロセスを検知したり，攻撃者がロードしたカーネルモジュールを検知したりすることができる．

IDS を従来のように VM 内に配置して監視を行うと，攻撃者が VM 内に侵入した際に IDS が無効化され，それ以降の検知を正常に行えなくなる恐れがある．そのため，VM の外側で IDS を動作させて VM の監視を行う IDS オフロード [5] が用いられている．VM 外にオフロードされた IDS は監視対象 VM のメモリ上の OS データを解析することで VM の監視を行う．しかし，VM のメモリが SEV の保護により暗号化されている場合，VM 外で動作する IDS は VM の監視を行うことができない．

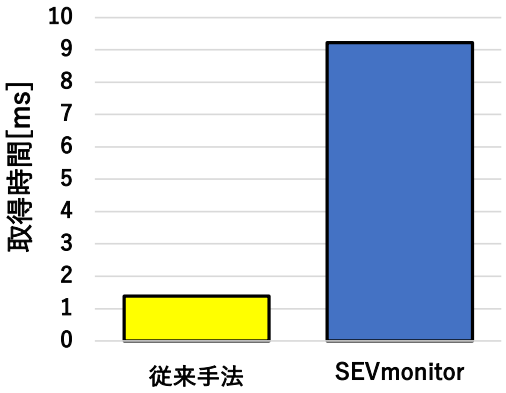

図 1 SEVmonitor のプロセス情報の取得時間

そこで，VM 外にオフロードした IDS が VM 内で安全に動作するエージェントと通信することでメモリデータを取得し，OS データの監視を行う SEVmonitor [2] が提案されている．SEVmonitor は VM 内で監視対象システムをコンテナまたは内部 VM に隔離することでエージェントを安全に実行する．これにより，SEV で暗号化された VM の監視を VM 外部から行うことができる．さらに，IDS はエージェントとの通信を暗号化することでメモリデータを安全に取得する．また，オフロードされた IDS も SEV で保護された別の VM で動作させることで，IDS が取得したメモリデータから機密情報が漏洩することを防ぐ．

SEVmonitor はページ単位でメモリデータを取得し，キャッシュに保持することで IDS とエージェント間の通信回数を削減している．このキャッシュにより，IDS が 1 つの構造体の中の複数のデータにアクセスする場合にはエージェントとの通信を行う必要がなくなる可能性が高い．しかし，プロセスリストなどのようにポインタを用いる OS データの場合，ポインタをたどるたびに通信が必要となることが多い．その結果，通信のオーバヘッドにより監視性能が大幅に低下する．オフロードした IDS が SEV で保護されていない VM のメモリを直接監視する従来手法と SEVmonitor の性能を比較する実験を行ったところ，図 1 のように全プロセスの情報取得にかかる時間が 6.6 倍に増加することが分かった．

3. eBPFmonitor

本稿では，SEV で保護された VM に eBPF プログラムを送り込み，OS データを先読みして一括で取得するシステム eBPFmonitor を提案する．eBPFmonitor のシステム構成を図 2 に示す．eBPFmonitor は SEVmonitor と同様に，監視対象 VM 内で監視対象システムをコンテナに隔離し，その外側で安全にエージェントを動作させる．IDS はエージェント経由で監視対象 VM のメモリデータをページ単位で取得する．それに加えて，監視対象 VM 内で eBPF プロ

© 1959 Information Processing Society of Japan

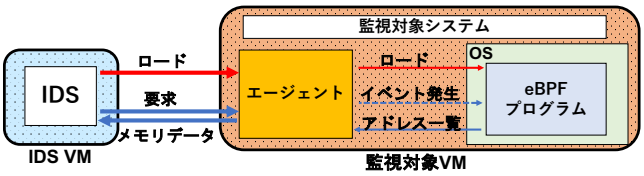

図 2 eBPFmonitor のシステム構成

グラムを実行することにより，監視に必要なメモリデータを一括で取得することができる．eBPF プログラムはポインタを用いる OS データを先読みし，一連の OS データを収集する．対象となる OS データの例としては，プロセスやカーネルモジュールのリスト，ネットワークソケットを管理するハッシュ表などが考えられる．このように，IDS からの 1 回の要求でエージェントが OS データを一括で返送することにより，通信のオーバヘッドを削減し，VM の監視を高速化する．

eBPFmonitor は OS データの収集に eBPF プログラムを用いることにより，安全性と柔軟性を両立させている．eBPF は性能などを監視するために用いられている Linux の機構であり，ネットワークやシステム動作のトレーシング，システムのセキュリティ対策などに使用されている．eBPF プログラムは OS 内にロードされ，ネットワークイベントやシステムコール，関数エントリ，カーネルのトレースポイントなどのイベントが発生した時に実行される．eBPF プログラムは OS の変更やカーネルモジュールの追加なしに OS 内に動的にロードして実行することができる．そのため，VM の監視に必要な OS データを IDS ごとに異なった eBPF プログラムを用いて収集することで，OS データの柔軟かつ効率的な一括取得を行うことができる．また，eBPF プログラムのロード時に検証器によって OS の実行に影響する命令や無限ループが検知されるとロードに失敗する．そのため，関数やグローバル変数へのアクセスが制限されるだけのカーネルモジュールより安全に OS データの取得を行うことができる．

eBPFmonitor では，IDS が VM の監視を開始する前にあらかじめ eBPF プログラムを監視対象 VM にロードしておく．IDS からエージェントを介して監視対象 VM の OS に eBPF プログラムをロードし，その際に eBPF プログラムを呼び出すためのイベントを OS に設定する．その後，IDS が OS データを必要とした時には IDS からエージェントに OS データの一括取得の要求を送信し，要求を受信したエージェントは eBPF プログラムを実行するためのイベントを発生させる．実行された eBPF プログラムはポインタをたどり OS データが格納されているアドレスを収集する．エージェントは収集されたアドレスに対するメモリデータを取得し，順次 IDS に返送する．IDS は受信したメモリデータをキャッシュに保存して利用する．

eBPF プログラムで OS データをどこまで一括取得するかにはトレードオフがある．極端なケースとして，IDS が必要とする可能性があるすべての OS データを一度に取得する方法が考えられる．これにより IDS からの要求回数は常に 1 回に削減されるが，取得した OS データの中に使われないデータがあった場合は，取得にかかった eBPF プログラムの実行時間とメモリデータの転送時間が無駄になる．例えば，IDS がリストをたどって目的のノードを探す場合，目的のノード以外では詳細なデータは使われない可能性が高い．もう一つの極端なケースは，SEVmonitor のように IDS が必要としたデータだけを要求する方法である．この方法では取得したデータの無駄はなくなるが，要求回数が増えるという問題がある．そのため，取得したデータの無駄を減らすために，ある程度の OS データだけを一括で取得するのが望ましい．

また，eBPF プログラムで収集する OS データの粒度にもトレードオフが存在する．eBPFmonitor は一括で OS データを取得しない場合には SEVmonitor と同様に，メモリデータをページ単位で取得する．そのため，一括取得する場合もページ単位でメモリデータを取得すると，IDS でのキャッシュ管理を容易にすることができる．しかし，ページに含まれる多くの OS データが IDS によって使われない場合にはメモリデータの転送時間が無駄になる．一方，eBPF プログラムを用いて OS データを収集する場合には IDS が必要とするデータを細かく指定することができるため，エージェントは必要なメモリデータだけを取得して返送することもできる．これにより，エージェントと IDS との通信データ量を削減し，取得したデータの無駄を減らすことができる．しかし，IDS でのキャッシュ管理が複雑になり，オーバヘッドが増大する可能性がある．また，eBPF プログラムが複雑になることにより，実行時間が長くなる可能性もある．

4. 実装

eBPFmonitor を BPF CO-RE [3] と LLView [4] を用いて Linux と KVM に実装した．

4.1 OS データを収集する eBPF プログラム

eBPFmonitor は eBPF プログラムを監視対象 VM に送り込むため，IDS を実行する VM と eBPF プログラムを実行する VM が異なるバージョンの Linux カーネルで動作していることがある．その場合，IDS 側でコンパイルした eBPF プログラムは監視対象 VM では動作しない可能性がある．BPF Compiler Collection (BCC) [6] を用いて監視対象 VM で実行時に eBPF プログラムがコンパイルされるようにすればこの問題は解決できるが，監視対象 VM に clang，LLVM，Linux のカーネルヘッダなどの eBPF プログラムの開発環境が必要になる．

そこで，eBPFmonitor は eBPF プログラムの作成と実

© 1959 Information Processing Society of Japan

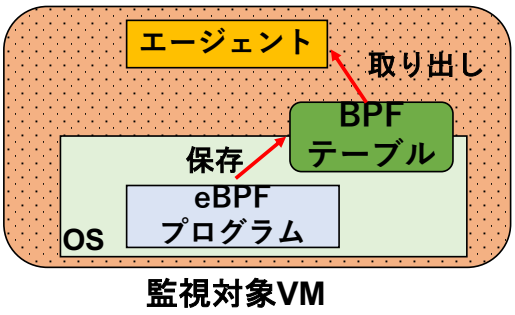

図 3 BPF テーブルを用いたアドレスの受け渡し

```
tasksoff = offsetof(struct task_struct, tasks);
pidoff = offsetof(struct task_struct, pid);

for (i=0; i<1000; i++) {
  task = next_task(task);
  addr = (unsigned long)BPF_CORE_READ(task, tasks.
      prev) - tasksoff + pidoff;
  addr &= PAGE_MASK;
  bpf_ringbuf_output(&rb, &addr, sizeof(addr), 0);
  if (BPF_CORE_READ(task, pid) == 0)
    break;
}
```

図 4 プロセスリスト用の eBPF プログラム

行に BPF Compile Once - Run Everywhere (CO-RE) と呼ばれるフレームワークを用いる．BPF CO-RE は IDS 側でコンパイルした eBPF プログラムを監視対象 VM でロードして実行することができる．OS のバージョンによる構造体の定義の違いを吸収するために，BPF CO-RE は eBPF プログラムのロード時に構造体のメンバ変数のオフセットの再配置を行う．この再配置を可能にするために，OS カーネルの構造体のメンバ変数へのアクセスには BPF_CORE_READ という BPF CO-RE が提供している関数を用いる．

ポインタをたどって OS データを収集するプログラムは一般的に終了条件付きのループとなるため，無限ループに陥る可能性がある．終了しない可能性のあるループを含む eBPF プログラムは OS のリソースの浪費を防ぐためにロード時の検証に失敗する．そのため，eBPFmonitor で用いる eBPF プログラムでは回数を指定した有限ループを用いる．また，eBPF には 1 度の実行で 100 万命令までしか使用することができないという制限もあるため，一回の実行ですべての OS データを収集できない場合がある．その場合には eBPF プログラムを複数回実行することで必要な OS データを収集する．

eBPFmonitor で用いる eBPF プログラムはポインタを用いるデータ構造をたどって OS データのアドレスを収集する．eBPF プログラムが収集したアドレスは図 3 のように BPF テーブルに格納され，エージェントに受け渡される．BPF テーブルはカーネル空間で動作する eBPF プログラムとユーザ空間のアプリケーションの間でデータを共有するために用いられるリングバッファである．BPF テーブルを用いることで，ユーザ空間のエージェントに高速にアドレスを受け渡すことができる．

最終的に，エージェントは収集したアドレスに対応するメモリデータを取得して IDS に返送するため，eBPF プログラムが直接メモリデータを BPF テーブルに格納することも考えられる．しかし，メモリデータをコピーするのにかなりの数の命令が使われ，メモリデータを暗号化する場合にはさらに非常に多くの命令が必要となる．そのため，命令数に制限のある eBPF プログラムでメモリデータを取得するのは難しい．また，eBPF プログラムで大量の命令を実行することによるオーバヘッドも無視できない．

我々は BPF CO-RE を用いてプロセスリストとカーネルモジュールリストをたどりアドレスを収集する eBPF プログラムを実装した．プロセスリスト用の eBPF プログラムは task_struct 構造体のリストをたどり，プロセス ID が格納されている pid メンバ変数とプロセス名が格納されている comm メンバ変数のアドレスを収集する．まず，eBPF のヘルパ関数である bpf_get_current_task 関数を使って現在のプロセスの task_struct 構造体のアドレスを取得し，そこからポインタをたどりプロセスリストの先頭を探す．そして，プロセスリストの先頭からメンバ変数のアドレスを収集する．pid メンバ変数のアドレスを収集する eBPF プログラムを図 4 に示す．このプログラムは 1000 回の有限ループになっており，1 回の実行で 1000 個までのプロセスの情報を収集できる．通常，&task->pid のように記述することで task_struct 構造体の pid メンバ変数のアドレスを取得できるが，BPF CO-RE ではこのような記述はできない．そこで，アドレスが格納されている tasks.prev メンバ変数の値を読み込み，そのメンバ変数のオフセットを引いて task_struct 構造体の先頭アドレスを計算する．そして，pid メンバ変数のオフセットを足すことで pid メンバ変数のアドレスを取得する．取得したアドレスからページの先頭アドレスを計算し，bpf_ringbuf_output 関数で BPF テーブルに格納する．

カーネルモジュールリスト用の eBPF プログラムは同様に module 構造体のリストをたどり，モジュール名が格納されている name メンバ変数のアドレスを収集する．プロセスリストと異なり，module 構造体を探すための eBPF のヘルパ関数は提供されていないため，カーネルモジュールリストの先頭アドレスを直接指定することでリストをたどっている．このアドレスはカーネルのバージョンに依存するため，IDS から先頭アドレスを渡せるようにすることが考えられる．また，現在のプロセスの task_struct 構造体から module 構造体の一つを探せるようにすることも考えられる．

© 1959 Information Processing Society of Japan

4.2 エージェント

IDS から eBPF プログラムのロードコマンドを受信すると，送られてきた eBPF プログラムを OS にロードする．次に，eBPF プログラムの実行を制御するためのフックポイントを設定する．複数の eBPF プログラムを使い分けられるようにするために，エージェントに用意した空の関数の一つをフックポイントとして設定する．そのために，エージェントの実行ファイルのパス名と関数名を指定して，ユーザ空間の動的ポイントをフックする Uprobe を用いる．そして，eBPF プログラムからのデータを受け取るための BPF テーブルを作成し，eBPF プログラムに割り振った番号を IDS に返す．

IDS から OS データの一括取得要求を受け取った際に，エージェントは eBPF プログラムを実行する．エージェントが受信した eBPF プログラムの番号に対応する関数を実行することで，OS 内にロードした eBPF プログラムが実行され，OS データのアドレスの収集が開始される．エージェントは BPF テーブルにアドレスが格納されるのを待ち，eBPF プログラムがアドレスを格納するとそれを読み出す．

eBPF プログラムが収集したアドレスを基に，エージェントは/dev/ekmem からメモリデータを取得する．/dev/ekmem はカーネルの仮想アドレスに対応するメモリデータを暗号化して返す疑似デバイスである．pread システムコールでページ番号をオフセットとして指定することで，そのページ番号に対応する 1 ページ分のメモリデータを返す．現在のところ，まだ暗号化は実装していない．/dev/ekmem を提供するためにカーネルモジュールの追加が必要となるが，/dev/ekmem は単純な実装であるため OS の信頼性は低下しないと考えられる．

4.3 eBPFmonitor ライブラリ

eBPFmonitor は IDS に対して，OS データをエージェントから透過的に取得するためのライブラリを提供する．このライブラリは初期化時にエージェントとのネットワーク接続を確立し，OS データをキャッシュするためのハッシュ表を作成する．IDS は VM の監視を開始する前にあらかじめ OS データの一括取得を行うための eBPF プログラムをライブラリに登録する．ライブラリはその eBPF プログラムを指定したロードコマンドをエージェントに対して送信する．エージェントが eBPF プログラムのロードに成功すると，ロードした eBPF プログラムに割り振られた番号が返される．

IDS は VM の監視に OS データを必要とした際にライブラリを呼び出す．この呼び出しは LLView フレームワーク [4] によって自動的に行われる．LLView は IDS プログラムのコンパイル時に LLVM の中間表現を出力し，中間表現の中の load 命令の前にライブラリを呼び出すコードを挿入する．呼び出されたライブラリはエージェントと通信し，IDS がアクセスしようとした OS データのアドレスを送信して対応するメモリデータを取得する．

OS データを一括取得する際には，ライブラリはエージェントに一括取得要求コマンドを送信する．このコマンドには一括取得する OS データに対応する eBPF プログラムに割り振られた番号と取得する OS データのアドレスを指定する．エージェントからメモリデータとそのメモリデータが格納されているページ番号の組が返されるため，ページ番号をキーとしてメモリデータをキャッシュに格納する．メモリデータとページ番号の組を繰り返し受信し，エージェントから終了を表すデータを受信すると一括取得を終了する．

OS データの一括取得を開始するタイミングとして様々なものが考えられる．例えば，リストの先頭ノードにアクセスした時や，リストのノードをたどった時などである．ライブラリは OS のデータ構造に関する情報を持たないため，IDS が一括取得を開始するアドレスやアドレス範囲をライブラリに登録する必要がある．現在のところ，ライブラリが最初に OS データを取得する際に一括取得を行うように実装している．

5. 実験

eBPFmonitor を用いて OS データを一括で取得できることを確認し，一括取得にかかる時間を調べた．実験に用いたマシンの CPU は Intel Core i7-10700，メモリは 64GB であった．ホスト OS として Linux 5.15.0，仮想化ソフトウェアとして QEMU-KVM 7.1.0 を動作させた．監視対象 VM には仮想 CPU を 2 個，メモリを 2GB 割り当て，ゲスト OS として Linux 5.15.0 を動作させた．本実験では SEV を用いた監視対象 VM の保護は行わなかった．

5.1 プロセス一覧の取得

eBPFmonitor を用いて監視対象 VM のプロセス一覧を取得する IDS を実行した．この IDS はプロセスリストをたどり各プロセスの ID と名前を収集する eBPF プログラムを用いた．IDS を実行した結果，219 ページ分のメモリデータを受信し，182 個のプロセス ID とプロセス名を出力した．出力結果から監視対象 VM 内の全プロセスの情報が取得できていることを確認した．また，eBPF プログラムを用いることで IDS からエージェントへの 1 回の要求でプロセス一覧が取得できることも確認した．

次に，eBPFmonitor を用いて監視対象 VM のプロセス一覧を取得する時間を測定した．IDS は起動時にエージェントとのネットワーク接続を確立し，エージェント経由で eBPF プログラムをロードする．その後も接続を維持し，定期的に OS データを取得して監視を行う．そのため，エージェントに OS データの一括取得要求を送信してから，

© 1959 Information Processing Society of Japan

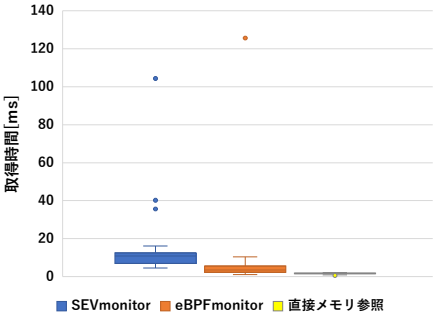

図 5　プロセス一覧の取得時間

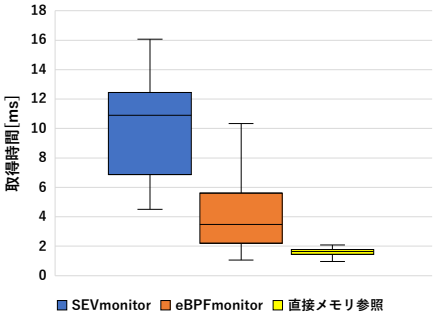

図 6　プロセス一覧の取得時間（外れ値を除去）

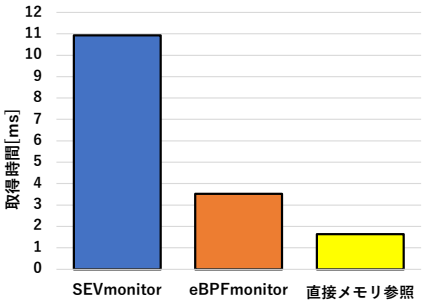

図 7　プロセス一覧の取得時間（中央値）

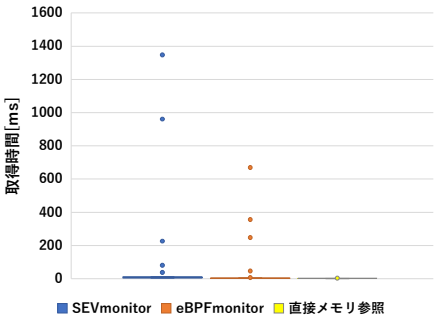

図 8　カーネルモジュール一覧の取得時間

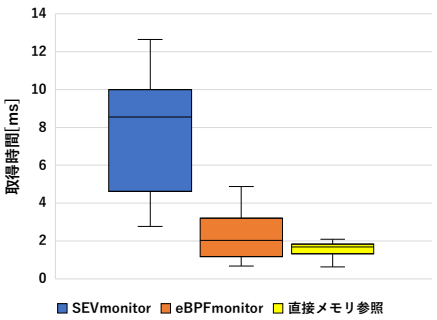

図 9　カーネルモジュール一覧の取得時間（外れ値を除去）

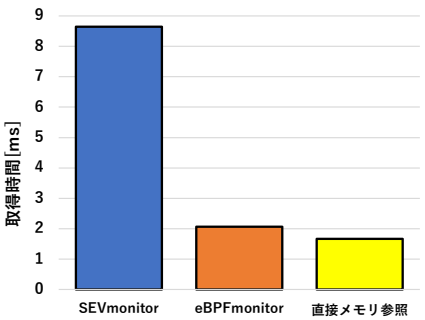

図 10　カーネルモジュール一覧の取得時間（中央値）

IDS がメモリデータを取得し，プロセス情報を出力するまでの時間を測定した．比較として，ポインタをたどるたびに通信を行う SEVmonitor と，エージェントを用いずに直接 VM のメモリを参照する従来手法でも取得時間の測定を行った．eBPFmonitor と SEVmonitor ではエージェントの分だけプロセス数が増えるため，この実験では取得するプロセスを 100 個に統一して取得時間の比較を行った．

100 回測定を行った時の取得時間を図 5 に示す．外れ値が多いため，外れ値を除去したグラフを図 6 に示す．さらに，3 つの手法を比較しやすくするために図 7 に中央値のみを示す．

図 7 より，eBPFmonitor は SEVmonitor より 3.1 倍高速にプロセス一覧を取得できることが分かった．SEVmonitor において IDS とエージェントは 118 回の往復通信を行ったのに対し，eBPFmonitor の通信は要求が 1 回，応答が 118 回となった．そのため，通信のオーバヘッドが削減され取得時間が減少した．一方，直接メモリ参照と比べると eBPFmonitor は 2.1 倍の取得時間となった．これは通信のオーバヘッドが主な原因と考えられる．ただし，SEV で保護された VM に対しては直接メモリ参照を行うことはできない．

図 5 より，SEVmonitor と eBPFmonitor において極端な外れ値が発生した一方，直接メモリ参照では発生しなかったことが分かる．そのため，外れ値は IDS とエージェントの通信によるものだと考えられるが，正確な原因については現在調査中である．図 6 より，eBPFmonitor の取得時間のばらつきが小さくなっていることが分かる．

5.2　カーネルモジュール一覧の取得

同様に，eBPFmonitor を用いて監視対象 VM のカーネルモジュール一覧を取得する IDS を実行した．この IDS はカーネルモジュールのリストをたどりモジュール名を収集する eBPF プログラムを用いた．IDS を実行した結果，77 ページ分のメモリデータを受信し，75 個のカーネルモジュール名を出力した．出力結果からリストの先頭アドレスを直接指定してリストをたどる eBPF プログラムでも，OS データを一括で取得できることを確認した．

次に，カーネルモジュール一覧の取得にかかる時間を測定した．プロセス一覧の取得と同様にして測定を行った．どの手法でも監視対象 VM で実行中のカーネルモジュールは変わらないため，全カーネルモジュールの情報を取得する時間を測定した．100 回測定を行った時の取得時間を図 8，図 9，図 10 に示す．

図 10 より，eBPFmonitor は SEVmonitor より 4.2 倍高速にカーネルモジュール一覧を取得できることが分かった．また，直接メモリ参照と比べると 1.2 倍の取得時間となった．eBPFmonitor と SEVmonitor の取得時間はプロセス一覧の取得時より減少し，直接メモリ参照の取得時間はほぼ同じであった．プロセス一覧の取得時と比べて通信データ量が少ないため，eBPFmonitor と SEVmonitor の

© 1959 Information Processing Society of Japan

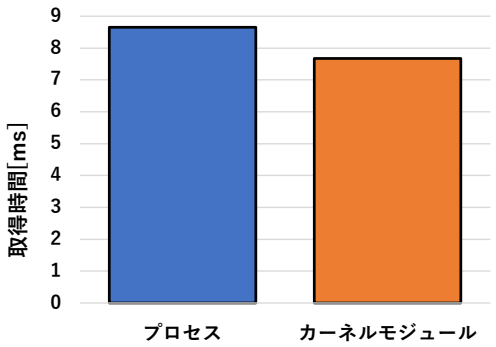

図 11 eBPF プログラムのロード時間

取得時間が減少したと考えられる．一方，プロセス一覧の取得時と比べて取得時間の減少率は SEVmonitor より eBPFmonitor の方が大きかった．これはカーネルモジュール一覧の取得に用いた eBPF プログラムではリストの先頭アドレスを探索する処理を行っていないため，eBPF プログラムの実行時間がプロセス一覧の取得時間より短くなったためと考えられる．

5.3 eBPF プログラムのロード

eBPF プログラムのロードにかかる時間を測定した．この実験では，IDS がエージェントとの通信を確立してから eBPF プログラムを VM 内の OS にロードし，IDS がエージェントからの返信を受け取るまでの時間を測定した．10 回測定を行い算出した平均値を図 11 に示す．プロセスリスト用の eBPF プログラムをロードする時間の方が少しだけ長いことが分かる．これはこの eBPF プログラムの方が少し長く，検証に時間がかかったためだと考えられる．

6. 関連研究

リモートホストのメモリを利用するシステムでデータ構造を意識した先読みが行われている．AIFM [7] はリモートホストのメモリをスワップ領域として用いるアプリケーションに専用のデータ構造を提供することで効率のよい先読みを可能にしている．例えば，リストの場合にはどちらの方向にたどっているかという情報を AIFM ランタイムが利用することで，同じ方向にあるオブジェクトを先読みする．AIFM はオブジェクト単位でスワップや先読みを行うことができる．AIFM では，専用のデータ構造を用いてアプリケーションを記述する必要があるため，既存のデータ構造を扱う OS データの監視には適用することができない．

DiLOS [8] はページング発生時にリモートホストのメモリデータを高速に取得することを可能にしている．ページングの際に発生するモード切り替えのオーバヘッドを Unikernel を用いることで削減している．さらに，アプリケーションの情報を利用してメモリデータの先読みを行う．ページフォールトが発生した時にアプリケーションを呼び出し，アプリケーションが今後，必要となるデータを指定することにより先読みを行う．その際に，ページよりも小さいサブページ単位でデータを先読みすることにより先読みの高速化を行う．送り込んだ eBPF プログラムが必要なデータを一括取得する eBPFmonitor と違い，ポインタをたどる際にはアプリケーションが必要なデータを一つずつ指定して先読みする必要がある．

BPFd [9] は BCC による eBPF プログラムのロードやフックポイントの設定などをリモートホストに転送することにより，リモートホストで eBPF プログラムを実行することができる．BCC による実行時の eBPF プログラムのコンパイルはローカルホストで行われるため，リモートホストとのカーネルバージョンの違いの影響を受ける可能性がある．一方，TeleBPF [10] は eBPF 関連のシステムコールをリモートホストに転送するため，BPF CO-RE と併用することによりカーネルバージョンの異なるリモートホストでも eBPF プログラムを正常に実行することができる．TeleBPF を eBPFmonitor で利用することも考えられるが，IDS が BPF テーブルに格納されたアドレスを取得した後，対応するメモリデータを取得するためにエージェントと逐一，通信する必要がある．

SEVmonitor [2] はエージェントを安全に動作させるために，エージェントを VM の中の OS 内またはハイパーバイザ内で動作させる．エージェントを OS 内で動作させる場合，OS が Linux であれば eBPF を利用することができるが，eBPF プログラムのロードや BPF テーブルへのアクセスを OS 内で行えるようにする必要がある．一方，エージェントをハイパーバイザ内で動作させる場合はハイパーバイザが KVM なら Linux の eBPF を利用することができる．しかし，ハイパーバイザ内の eBPF プログラムが内部 VM の OS データを取得できるようにする必要がある．この実装にはハイパーバイザ内で eBPF プログラムを実行して VM の OS データを取得する Hyperupcalls [11] を用いることができる．

7. まとめ

本稿では，SEV で保護された VM に送り込んだ eBPF プログラムを用いて OS データを先読みして一括で取得するシステム eBPFmonitor を提案した．eBPFmonitor は eBPF プログラムを用いることで，先行研究より通信の回数を減らし高速に OS データを取得することができる．実験結果より，IDS から監視対象 VM への 1 回の要求で，必要とするすべてのメモリデータを一括で取得できることを確認し，それにより取得時間を大幅に高速化できることが分かった．

今後の課題は，IDS が必要とするデータのみを一括で取得できるようにし，通信データ量の削減によるさらなる高速化を行うことである．また，VM に SEV を適用し，IDS

© 1959 Information Processing Society of Japan

とエージェントの通信を暗号化した時の性能を比較することも必要である．さらに，エージェントを OS 内やハイパーバイザ内で動作させられるようにすることでエージェントの保護を強化することも検討している．

謝辞 本研究の一部は，JST，CREST，JPMJCR21M4 の支援を受けたものである．また，本研究の一部は，国立研究開発法人情報通信研究機構の委託研究（05501）による成果を含む．

参考文献

[1] Advanced Micro Device: Secure Encrypted Virtualization API Version 0.24 (2020).
[2] 能野智玄，光来健一：AMD SEV で保護された VM の隔離エージェントを用いた安全な監視，*CSS 2022* (2022).
[3] Nakryiko, A.: BPF CO-RE (Compile Once - Run Everywhere), Andrii Nakryiko's Blog (online), available from ⟨https://nakryiko.com/posts/bpf-portability-and-co-re/⟩ (accessed 2023-4-10).
[4] Ozaki, Y., Kanamoto, S. and Kourai, K.: Detection System Failures with GPUs and LLVM, *Proc. Asia Pacific Workshop on Systems*, pp. 47–53 (2019).
[5] Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. Network and Distributed System Security Symposium* (2003).
[6] IO Visor Project: BPF Compiler Collection (BCC), GitHub (online), available from ⟨https://github.com/iovisor/bcc⟩ (accessed 2023-4-14).
[7] Ruan, Z., Schwarzkopf, M., Aguilera, M. K. and Belay, A.: AIFM: High-Performance, Application-Integrated Far Memory, *Proc. USENIX Symp. Operating Systems Design and Implementation*, pp. 315–332 (2020).
[8] Yoon, W., Oh, J., Ok, J., Moon, S. and Kwon, Y.: DiLOS: adding performance to paging-based memory disaggregation, *Proc. ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 70–78 (2021).
[9] Fernandes, J.: BPFd (Berkeley Packet Filter Daemon), GitHub (online), available from ⟨https://github.com/joelagnel/bpfd⟩ (accessed 2023-4-11).
[10] 堀恭介，光来健一：eBPF プログラムを送り込むことによる VM 内のシステム監視，*SWoPP* (2022).
[11] Amit, N. and Wei, M.: The Design and Implementation of Hyperupcalls, *Proc. USENIX Annual Technical Conf.*, pp. 97–112 (2018).

© 1959 Information Processing Society of Japan