eBPFを用いたConfidential VMの安全かつ高速な監視

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，最近のクラウドはメモリが暗号化された Confidential VM を提供することにより，VM 外からの盗聴を防げるようにしている．Confidential 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，Confidential VM に eBPF プログラムを送り込み，OS データを先読みすることにより VM の監視を高速化するシステム eBPFmonitor を提案する．eBPFmonitor はポインタを用いる OS データを一括で取得することにより，通信のオーバヘッドを削減する．eBPFmonitor は BPF CO-RE を用いることにより，eBPF プログラムを任意の監視対象 VM に送り込んで実行することを可能にする．また，LLVM Pass を用いて eBPF プログラムを変換することにより，OS データの収集を自動化する．eBPFmonitor を用いてプロセス，カーネルモジュール，TCP ソケットの一覧を一括取得する IDS を開発し，IDS の性能が最大 70%向上することが確認できた．

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，最近のクラウドは Confidential VM を提供することにより，VM 外からの盗聴を防げるようにしている．Confidential VM は AMD SEV [1] などの Trusted Execution Environment (TEE) を用いてメモリが暗号化された VM である．TEE が用いる暗号鍵は CPU によって管理されているため，クラウドの内部犯であってもメモリデータを復号することはできない．

一方，Confidential VM であっても 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，Confidential VM 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，Confidential VM を用いると VM の外に IDS をオフロードして VM のメモリ上の OS データを監視することはできなくなる．

そこで，IDS が VM 内で安全に動作するエージェントと通信して OS データを取得し，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，Confidential VM に対する侵入検知を VM 外部から行うことができる．しかし，監視に用いる OS 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，Confidential 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 LLVM Pass [4] を用いて実装した．eBPF プログラムの開発に BPF CO-RE を用いることにより，Linux のカーネルバージョンによる

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

© 1959 Information Processing Society of Japan

構造体の定義の違いを吸収する．それにより，開発環境でコンパイルした eBPF プログラムを任意の監視対象 VM でロードして実行することを可能にする．また，eBPF プログラムをコンパイルして生成された中間表現を LLVM Pass を用いて変換することにより，OS データの収集を自動化することができる．eBPFmonitor を用いて実験を行い，監視対象 VM のプロセス，カーネルモジュール，TCP ソケットの一覧を一括で取得できることを確認した．また，eBPFmonitor は SEVmonitor より監視の性能がプロセス一覧の取得で 59%，カーネルモジュール一覧の取得で 70%，TCP ソケット一覧の取得で 20%向上した．

以下，2 章では Confidential 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 Confidential 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し，5 章では eBPFmonitor の動作確認と性能測定のために行った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. Confidential VM の監視

近年，組織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，最近のクラウドは Confidential VM を提供し，内部犯による VM 内の機密情報の盗聴を防げるようにしている．Confidential VM は AMD SEV などの TEE 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．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はできない．

一方，Confidential VM のメモリは暗号化されているが，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，Confidential VM 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，Confidential VM による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，

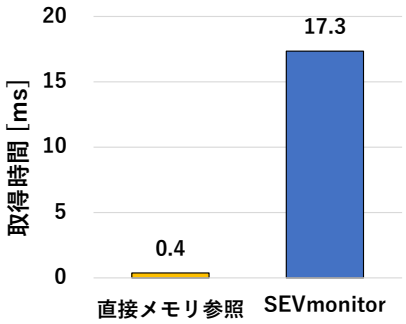

図 1 SEVmonitor のプロセス情報の取得時間

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，Confidential VM はメモリが暗号化されているため，VM 外で動作する IDS は VM の監視を行うことができない．

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，Confidential 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 も別の Confidential VM で動作させることにより，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は 1 回で済む可能性が高い．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 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 48 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

© 1959 Information Processing Society of Japan

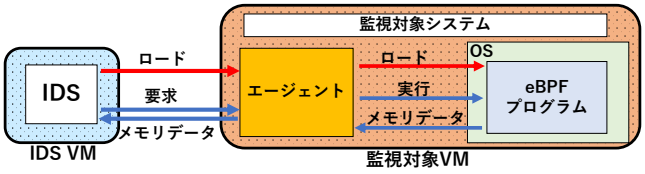

図 2 eBPFmonitor のシステム構成

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もしくはデータ単位で取得する．それに加えて，監視対象 VM 内で eBPF プロ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使用する eBPF プログラムが監視対象 VM にロードされているかどうかを確認する．eBPF プログラムのハッシュ値をエージェントに送信することで，確認を行う．該当する eBPF プログラムが監視対象 VM にロードされていなかった場合は，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が BPF 用のシステムコールを用いて eBPF プログラムを実行する．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やデータを収集する．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]，LLVM Pass [4]，LLView [6] 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

4.1.1 BPF CO-RE の利用

eBPFmonitor は eBPF プログラムを監視対象 VM に送り込むため，eBPF プログラムの開発環境と実行環境とで異なるバージョンの Linux カーネルが動作しているのが一般的である．Linux のデータ構造はカーネルのバージョンによって少しずつ異なるため，開発環境で動作している Linux のヘッダファイルを用いてコンパイルした eBPF プ

© 1959 Information Processing Society of Japan

ログラムは監視対象 VM では動作しない可能性がある．監視対象 VM の Linux のバージョンに合わせてコンパイルすることも考えられるが，Linux のバージョンが変わるたびにコンパイルし直す必要がある．この問題は BPF Compiler Collection (BCC) [7] を用いて監視対象 VM でのロード時に eBPF プログラムをコンパイルするようにすれば解決できるが，監視対象 VM に eBPF プログラムの開発環境が必要になり，ロード時に時間がかかる．

そこで，eBPFmonitor は eBPF プログラムの開発と実行に BPF Compile Once – Run Everywhere (CO-RE) フレームワーク [3] を用いる．BPF CO-RE は開発環境でコンパイルした eBPF プログラムを任意の監視対象 VM でロードして実行することができる．Linux のカーネル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．そのために，Linux カーネルの構造体のメンバ変数へのアクセスには `task->pid` のような通常の方法ではなく，BPF CO-RE が提供する `BPF_CORE_READ` マクロを用いる．また，ビットフィールドへのアクセスには `BPF_CORE_READ_BITFIELD_PROBED` マクロを用いる．これらのマクロでは構造体のメンバ変数に対して BPF の組み込み関数で再配置が行われ，その結果を用いてカーネルの構造体のメンバ変数の値を取得する．`BPF_CORE_READ_BITFIELD_PROBED` マクロは目的のビットフィールドを含むメモリデータをカーネルから取得し，その値に対してビット演算を行うことで目的の値を取得する．

再配置を行うための構造体の情報は BPF Type Format (BTF) [8] から取得する．BTF は eBPF プログラムのデバッグ情報を生成するためのデータ型や関数の情報が記述されたメタデータフォーマットである．eBPF プログラムのコンパイル時に BTF 情報として構造体のメンバ変数の名前とデータ型が組み込まれる．その BTF 情報を使うことで，実行先のカーネルの BTF から実行環境に適した構造体のメンバ変数のオフセットを取得できる．また，eBPF プログラムはカーネルのヘッダファイルの代わりに vmlinux.h を用いて開発を行う．vmlinux.h は Linux カーネルのデータ型が記述された BTF 情報が C 言語で扱えるデータ構造体に変換されたものである．vmlinux.h を用いて eBPF プログラムの開発を行うことにより，カーネルのバージョンに依存せずにカーネルの構造体の利用が可能となる．

BPF CO-RE を用いる場合には，eBPF プログラム内でカーネルのグローバル変数を直接使用することができない．`bpf_get_current_task` 関数などの BPF ヘルパー関数を用いることで代替可能なものもあるが，その種類は限られる．eBPF プログラムにグローバル変数のアドレスを埋め込む方法が考えられるが，このアドレスはカーネルのバージョンに依存し，ロードするカーネルで KASLR が有効になっていると事前に実行時のアドレスを知ることができない．そこで，BPF ヘルパー関数の `bpf_kallsyms_lookup_name` 関数を用いて実行時にグローバル変数の名前からアドレスに変換し，そのアドレスを用いる．これにより，eBPF プログラムをロードしたカーネルに合わせてグローバル変数にアクセスすることができる．ただし，開発環境で存在したグローバル変数が実行環境でなくなっていたり，名前が変わっていたりすると変換に失敗する．

```
static long callback_fn(u32 index, void *ctx) {
  ...
  return 0;
}

int proc_list(void *ctx) {
  bpf_loop(1000, callback_fn, NULL, 0);
}
```

図 3 `bpf_loop` 関数を用いた eBPF プログラム

4.1.2 ループ処理

ポインタをたどって OS データを収集するプログラムは終了条件を満たすまで処理を繰り返すことが多い．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 のリソースの消費を防ぐためにロード時の検証に失敗する．このような処理を行う場合には，最大回数を指定した有限ループを用い，終了条件を満たした時にループを終了するようにする方法が考えられる．しかし，有限ループはコンパイル時に展開されるため，ループ内での処理が増えたり多重ループを用いたりすると命令数が増加し，一回の実行で 100 万命令までしか使用することのできないという eBPF の制限をすぐに超えてしまう．

そこで，BPF ヘルパー関数の `bpf_loop` を用いることで，ループ可能な回数を大幅に増やした有限ループを用いることができる．このヘルパー関数は最大で 800 万回のループが可能であるため，OS データを収集するのに十分な回数を指定することができる．また，ループが展開されないため，多重ループを用いても eBPF の制限を超えにくい．`bpf_loop` 関数を用いる場合にはループの中身をコールバック関数として記述し，ループを途中で終了する場合には 1 を返す．コールバック関数にまたがって利用したい変数は `bpf_loop` 関数の引数に指定することで，コールバック関数の引数として渡される．しかし，その変数を用いた if 文が正常に実行されないという問題が発生した．eBPF プログラムの JIT コンパイル時に if 文で初回に実行された部分だけが残り，それ以外の条件文とその中身は削除される．この問題を回避するために，コールバック関数にまたがる値の共有にはグローバル変数を用いる．`bpf_loop` 関数を用いた eBPF プログラムの例を図 3 に示す．

有限ループを用いる方法以外に，BPF Iterator [9] という仕組みを用いることも可能である．BPF Iterator は eBPF

© 1959 Information Processing Society of Japan

```
SEC("iter/task")
int proc_list(struct bpf_iter__task *ctx) {
  struct task_struct *task = ctx->task;
  ...
}
```

図 4 BPF Iterator を用いた eBPF プログラム

```
struct bpf_iter_num it;
int *i;
...
bpf_iter_num_new(&it, 0, 1000);
while ((i = bpf_iter_num_next(&it))) {
  ...
}
bpf_iter_num_destroy(&it);
```

図 5 BPF KFuncs を用いた eBPF プログラム

プログラムの 1 つのタイプであり，カーネルのデータ構造体のリストをたどりながら指定されたコールバック関数を呼び出す．bpf_loop を用いる方法と比較して，リストをたどる処理を記述する必要がないため eBPF プログラムの作成が容易になる．しかし，BPF Iterator は対応しているカーネルのデータ構造体に対してしか利用することができない．BPF Iterator を用いた eBPF プログラムの例を図 4 に示す．

他にも BPF Kernel Functions (KFuncs) [10] と呼ばれる関数を用いて，eBPF プログラム内でループ処理を行うことが可能である．BPF KFuncs は Linux カーネルの関数を eBPF プログラム内で使用できるようにした関数である．有限ループを行うための関数やカーネルのデータ構造体のリストをたどる関数が提供されており，コールバック関数を用いずに eBPF プログラムを記述することができる．しかし，BPF KFuncs は BPF ヘルパー関数や BPF プログラムタイプと違い，Linux カーネルのバージョンによって関数が変更されたり削除されたりする可能性がある．BPF KFuncs の有限ループを行う関数を用いた eBPF プログラムの例を図 5 に示す．

4.1.3 OS データの収集

eBPF プログラムはカーネルのデータ構造に含まれるポインタをたどりながら先読みする OS データを収集する．収集した OS データは bpf_ringbuf_reserve ヘルパー関数と bpf_ringbuf_submit ヘルパー関数を用いて図 6 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 OS データを受け渡すことができる．

OS データをデータ単位で収集する場合，eBPF プログラムは OS データのアドレス，サイズ，値を BPF テーブルに格納する．この場合，グローバル変数，構造体メンバ，配列の要素，文字列などの粒度で収集を行う．文字列を収集する場合は文字列が格納されている配列のサイズを文字列の長さとすることで，文字列を BPF テーブルに格納している．一方，メモリページ単位で収集を行う場合，eBPF プログラムは OS データに含まれるページの番号だけを BPF テーブルに格納する．これはメモリデータそのものを BPF テーブルに格納すると多くの eBPF 命令が必要となり，実行オーバヘッドも大きくなるためである．

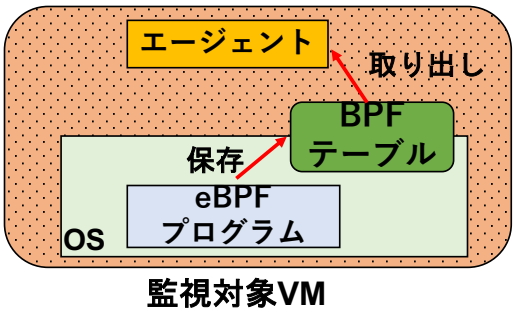

図 6 BPF テーブルを用いた OS データの受け渡し

eBPF プログラムにおいて OS データを収集する処理は Linux カーネルのソースコードを利用して実装することができる．OS データへのアクセスを行っている箇所に処理を追加して，BPF テーブルに OS データを格納する．しかし，複雑なデータ構造を用いて管理されている OS データの場合，OS データへのアクセスは様々な箇所で行われる．それら一つ一つに対して手動で BPF テーブルに OS データを格納する処理を追加するのは骨が折れる作業である．

開発者の負荷を減らすために，eBPFmonitor は eBPF プログラムがアクセスした OS データを自動的に BPF テーブルに格納する．そのために，LLVM を用いて eBPF プログラムをコンパイルし，生成された中間表現を Pass を用いて変換する．Pass は中間表現に対して最適化を施すための LLVM の仕組みである．eBPFmonitor は中間表現の中の bpf_probe_read_kernel 関数を呼び出す箇所に OS データを BPF テーブルに格納する関数の呼び出しを挿入することで，OS データを BPF テーブルに格納する．bpf_probe_read_kernel 関数は BPF CO-RE を用いて開発された eBPF プログラムがカーネルから OS データを読み込む際に呼び出される．OS データのアドレスやサイズの情報はこの関数の引数から取得する．

4.1.4 eBPF プログラムの例

eBPFmonitor を用いてプロセス，カーネルモジュール，TCP ソケットの一覧を先読みする eBPF プログラムを開発した．プロセス一覧を先読みする eBPF プログラムは task_struct 構造体のリストをたどり，pid メンバに格納されているプロセス ID の値，comm メンバに格納されているプロセス名の文字列，次のリストノードを指すポイ

© 1959 Information Processing Society of Japan

```
static struct task_struct *task;
static u64 init = 0;

static long callback_fn(u32 index, void *ctx) {
  int pid = BPF_CORE_READ(task, pid);

  task = next_task(task);
  if ((u64)task == init)
    return 1;
  return 0;
}

SEC("syscall")
int proc_list(void *ctx) {
  char name[10] = "init_task";

  bpf_kallsyms_lookup_name(name, sizeof(name),
       0, &init);
  task = (struct task_struct *)init;
  bpf_loop(1000, callback_fn, NULL, 0);

  return 0;
}
```

図 7 プロセスリスト用の eBPF プログラム

ンタの値を収集する．まず，bpf_kallsyms_lookup_name ヘルパー関数を用いて init_task というグローバル変数名からプロセスリストの先頭アドレスを取得する．そして，プロセスリストの先頭からポインタをたどりながら task_struct 構造体のメンバ変数の情報を収集し，プロセスリストの先頭に戻ってきたらループを終了する．この eBPF プログラムの一部を図 7 に示す．このプログラムは loop 関数を最大 1000 回実行し，1 回の実行で 1000 個までのプロセスの情報を収集する．

カーネルモジュール一覧を先読みする eBPF プログラムはプロセス一覧の先読みと同様に，module 構造体のリストをたどり，name メンバに格納されているモジュール名の文字列と次のリストノードを指すポインタの値を収集する．そのために，modules というグローバル変数名からカーネルモジュールリストの先頭アドレスを取得し，module 構造体のメンバ変数の情報を収集する．

TCP ソケット一覧を先読みする eBPF プログラムはネットワークソケットを管理するハッシュ表をたどり，/proc/net/tcp 疑似ファイルを生成する時と同様の情報を収集する．まず，tcp_hashinfo というグローバル変数名からハッシュ表のアドレスを取得する．そして，Linux カーネル内の proc ファイルシステムと同様の処理を行ってハッシュ表からネットワークソケットの情報を取得する．具体的には，tcp_seq_start 関数で初期値を取得し，tcp4_seq_show 関数でソケットの情報を取得して，tcp_seq_next 関数で次に処理するソケットを探索する．ただし，eBPF プログラムでは疑似ファイルを作成する必要はないため，その処理は行わないようにした．

4.2 エージェント

エージェントは IDS から eBPF プログラムのロード要求を受信すると，その eBPF プログラムが OS にロードされているかどうかを確認する．eBPF プログラムのロード状況はバイナリのハッシュ値を用いて管理する．IDS から送られてきたハッシュ値が未登録の場合は IDS に eBPF プログラムを要求し，受信した eBPF プログラムを OS にロードする．エージェントは eBPF プログラムのロードが完了すると，その結果を IDS に返送する．

BPF CO-RE は OS 内で実行される eBPF プログラムとそれを OS にロードし実行するユーザ空間のコードからなる一つのバイナリを生成する．しかし，ユーザ空間で実行されるプログラムは eBPF による検証が行われないため，監視対象 VM に送り込んで実行すると安全性が低下する．そのため，eBPFmonitor では eBPF プログラムのオブジェクトファイルだけを取り出して IDS からエージェントに送信する．ユーザ空間で実行されるコードは共通化してエージェントに統合しておくことで，安全性が保証されたプログラムだけを監視対象 VM で実行する．

エージェントは IDS から OS データの先読み要求を受け取ると，指定されたハッシュ値に対応する eBPF プログラムを実行する．eBPF プログラムは通常，フックポイントを設定することでイベント発生時に実行されるが，eBPFmonitor では bpf システムコールを用いることで直接実行する．

ページ単位で OS データの収集を行う場合，エージェントは eBPF プログラムが BPF テーブルに格納したページ番号を基に/dev/ekmem からその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．メモリのアドレスとサイズをバッファに格納して pread システムコールを図 8 のように呼び出すことで，そのアドレスに対応するサイズ分のメモリデータを暗号化して返す．データ単位で OS データを収集する場合も/dev/ekmem を使用できるように，取得するメモリデータのサイズを指定している．ページ単位の収集の場合は，OS データのアドレスとしてページ番号，サイズとしてメモリページのサイズが指定され，1 ページ分のメモリデータが返される．/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．

データ単位で OS データの収集を行う場合，eBPF プログラム内でメモリデータの取得を行うため，/dev/ekmem は使用しない．BPF テーブルに格納されたメモリデータは暗号化されていないため，エージェントが暗号化を行う．この方法ではエージェントが攻撃された場合にメモリデータを盗聴される可能性があるため，ページ単位の場合と同様に/dev/ekmem を用いて暗号化されたメモリデータ

© 1959 Information Processing Society of Japan

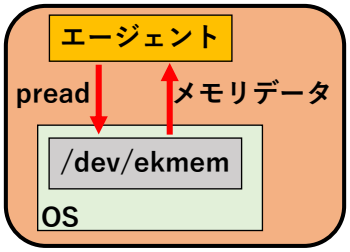

図 8 /dev/ekmem を用いたメモリデータの取得

を取得する方法や，eBPF プログラム内で暗号化を行う方法を検討している．eBPF プログラムが暗号アルゴリズムを実行すると非常に多くの命令数が必要となるため，BPF KFuncs として提供される暗号化を行う関数を用いることが考えられる．

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経由で透過的に取得するためのライブラリを提供する．このライブラリはエージェントとのネットワーク接続を確立し，一括取得した 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 から OS データの一括取得を行うための eBPF プログラムが登録されると，ライブラリはその eBPF プログラムのハッシュ値を計算し，エージェントに送信する．監視対象 VM でその eBPF プログラムがまだロードされていない場合にはエージェントに eBPF プログラムを送信する．

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 [6] 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 IDS がアクセスしようとした OS データがキャッシュに格納されているかどうか確認する．格納されていた場合は，キャッシュされた OS データを IDS に返す．格納されていない場合はエージェントと通信し，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．

その際に，ライブラリが OS データの一括取得を行う場合には，エージェントに一括取得要求を送信する．エージェントが対応する eBPF プログラムを実行して暗号化されたデータとそのサイズの組を返すと，ライブラリはそのデータを復号する．そして，OS データのアドレス，サイズ，メモリデータを取得し，アドレスをキーとしてメモリデータをキャッシュに格納する．ライブラリは一括取得の終了までエージェントから OS データを繰り返し受信して，同様の処理を行う．

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，IDS が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 eBPF プログラムのロードを行う際に一括取得を行っている．

5. 実験

eBPFmonitor を用いて IDS が OS データを一括で取得して正しく動作することを確認し，一括取得にかかる時間を調べた．この実験では，IDS ごとに必要な OS データを先読みする eBPF プログラムを用い，OS データをページ単位またはデータ単位で BPF テーブルに格納する処理を自動挿入した．比較のために，OS データのポインタをたどる度にページ単位でメモリデータを取得する SEVmonitor と，VM のメモリから直接，OS データを取得する従来手法を用いた．

実験に用いたマシンの CPU は AMD EPYC 7443P，メモリは 250GB 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 6.2.0 を動作させた．IDS VM と監視対象 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 6.5.0 を動作させた．IDS VM と監視対象 VM はどちらも SEV による保護を行った．ただし，従来手法を用いて直接メモリ参照を行う場合には，監視対象 VM の SEV による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 と名前を取得して出力する．この IDS を実行した結果，OS データをページ単位またはデータ単位で先読みした場合も SEVmonitor と同じ出力結果が得られることが分かった．また，IDS の出力結果と監視対象 VM 内での ps コマンドの実行結果を比較したところ，全プロセスの情報を正しく取得できていることが確認できた．

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て eBPF プログラムの登録を行い，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 eBPF プログラムのロードを行った後，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．監視を行っている最中に監視対象 VM で動作するプロセスの数が増減することがあるため，この実験では取得するプロセスを 150 個に統一して取得時間の比較を行った．

取得時間の測定を 100 回行った時の平均値を図 9 に示す．また，プロセス一覧を取得する際に行われた IDS から

© 1959 Information Processing Society of Japan

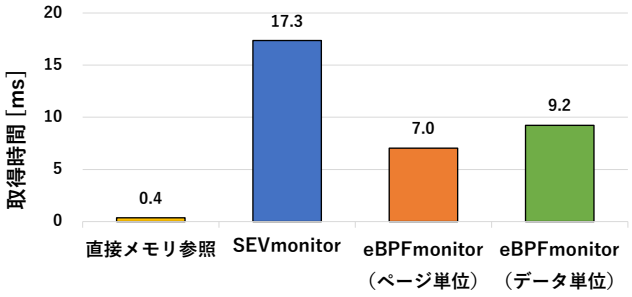

図 9 プロセス一覧の取得時間

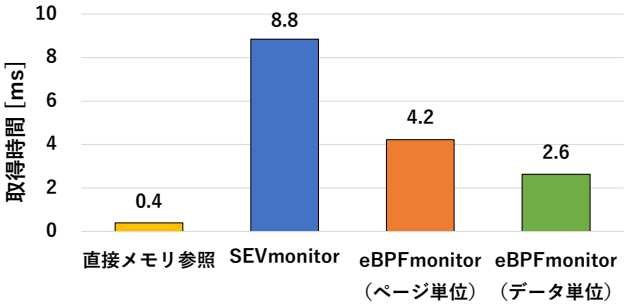

図 10 カーネルモジュール一覧の取得時間

表 1 プロセス一覧の取得における通信の詳細

	要求	応答	データ量 [KB]
SEVmonitor	150	150	615
eBPFmonitor（ページ単位）	1	151	617
eBPFmonitor（データ単位）	1	451	16

表 2 カーネルモジュール一覧の取得における通信の詳細

	要求	応答	データ量 [KB]
SEVmonitor	77	77	316
eBPFmonitor（ページ単位）	1	78	317
eBPFmonitor（データ単位）	1	155	9

エージェントへの OS データの要求回数，エージェントから IDS への OS データの応答回数，IDS が受信した OS データのデータ量を表 1 に示す．eBPFmonitor は SEVmonitor よりプロセス一覧の監視の性能が 47%〜59%向上した．これは OS データの要求回数が 1 回に減少して通信のオーバヘッドが削減され，取得時間が減少したためである．一方，直接メモリ参照と比べると eBPFmonitor は 20 倍の取得時間がかかった．これは通信のオーバヘッドと暗号化が主な原因と考えられる．ただし，Confidential VM に対しては直接メモリ参照を行うことはできない．

また，OS データをデータ単位で取得するとページ単位で取得するより 31%監視性能が低下したことが分かる．データ単位で取得すると通信データ量は大幅に減少したが，OS データの応答回数が大幅に増加したため監視性能が低下したと考えられる．さらに，OS データをデータ単位で取得する場合には eBPF プログラム内でメモリデータの取得まで行っている．そのため，eBPF プログラム内での処理の増加によるオーバヘッドが大きいことも原因として考えられる．

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，モジュール名を取得して出力する．この IDS を実行した結果，OS データをページ単位またはデータ単位で先読みした場合も SEVmonitor と同じ出力結果が得られることが分かった．また，IDS の出力結果と監視対象 VM 内での lsmod コマンドの実行結果を比較したところ，全カーネルモジュールの情報を正しく取得できていることが確認できた．

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．取得時間の測定を 100 回行った時の平均値を図 10 に示す．また，カーネルモジュール一覧を取得する際に行われた通信の詳細を表 2 に示す．eBPFmonitor は SEVmonitor よりカーネルモジュール一覧の監視の性能が 52%〜70%向上し，プロセス一覧の取得の場合よりも性能が向上した．これは IDS からエージェントへの 1 回の要求で必要な OS データがすべて取得でき，通信のオーバヘッドが削減されたためである．直接メモリ参照と比べても 6.8 倍の取得時間となり，プロセス一覧の取得より性能がよいことが分かった．

一方，プロセス一覧の取得とは異なり，OS データをデータ単位で取得するとページ単位で取得するより 38%監視性能が向上した．プロセス一覧の取得と同様に，データ単位で取得する場合，通信データ量は減少したが，OS データの応答回数は増加した．しかし，OS データの応答回数がプロセス一覧の取得では約 3 倍に増加したのに対して，カーネルモジュール一覧の取得では約 2 倍の増加であった．そのため，通信回数の増加によるオーバヘッドよりも通信データ量の削減による高速化の方が大きかったことが原因と考えられる．

5.3 TCP ソケット一覧の取得

同様に，eBPFmonitor を用いて監視対象 VM の TCP ソケット一覧を取得する IDS を実行した．この IDS は TCP ソケットのハッシュ表をたどり，通信元と通信先のアドレスとポート，通信の状態などを取得して出力する．この IDS を実行した結果，OS データをページ単位またはデータ単位で先読みした場合も SEVmonitor と同じ出力結果が得られることが分かった．また，IDS の出力結果と監視対象 VM 内の `/proc/net/tcp` 疑似ファイルを比較したところ，すべての TCP ソケットの情報を正しく取得できていることが確認できた．

次に，TCP ソケット一覧の取得にかかる時間を測定した．取得時間の測定を 100 回行った時の平均値を図 11 に示す．また，TCP ソケット一覧を取得する際に行われた

© 1959 Information Processing Society of Japan

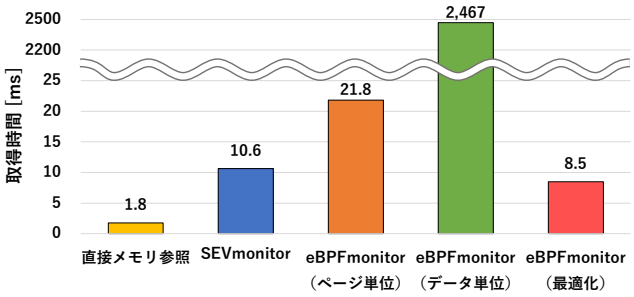

図 11 TCP ソケット一覧の取得時間

表 3 TCP ソケット一覧の取得における通信の詳細

	要求	応答	データ量 [KB]
SEVmonitor	68	68	279
eBPFmonitor（ページ単位）	7	76	309
eBPFmonitor（データ単位）	16	17696	702
eBPFmonitor（最適化）	1	74	300

IDS からエージェントへの通信の詳細を表 3 に示す．プロセス一覧やカーネルモジュール一覧の取得とは異なり，eBPFmonitor は SEVmonitor より TCP ソケット一覧の取得が 2〜232 倍に増加することが分かった．この IDS は 1 回の要求ですべての OS データを取得することができておらず，追加で 6〜15 回の要求を送る必要があったが，ここまでの性能低下につながるとは考えにくい．特に，ページ単位で取得する場合には通信回数とデータ量に大きな違いはなかった．

eBPF プログラムの処理を詳細に調べたところ，BPF テーブルに OS データを格納する回数が約 10 万回と非常に多くなっていることが分かった．これは eBPF プログラムが OS データにアクセスする度にその情報を BPF テーブルに格納するためである．ただし，その情報をエージェントにおいて重複しないように IDS に送信するため，通信量は増加しなかった．このように，BPF テーブルに格納する処理のオーバヘッドによって監視性能が低下したと考えられる．それに加えて，OS データをデータ単位で収集する場合，OS データごとにアドレスとサイズの情報を送信するため，ページ単位で収集する場合よりも通信データ量が大幅に増えた．そのため，データ単位で収集するメリットがなくなったと考えられる．

そこで，BPF テーブルに OS データを格納する処理を LLVM を用いず手動で挿入した eBPF プログラムを作成した．この eBPF プログラムは BPF テーブルに何度も格納されるページのアドレスを一度だけ格納するように最適化を行った．この IDS を実行した結果，TCP ソケット一覧を正しく取得できることを確認した．また，図 11 から SEVmonitor より取得時間が 20%減少したことが分かる．表 3 に示すように，1 回の要求で必要な OS データをすべて取得することができた．SEVmonitor より通信量は少し増えたが，通信のオーバヘッドが大幅に削減された．一方，

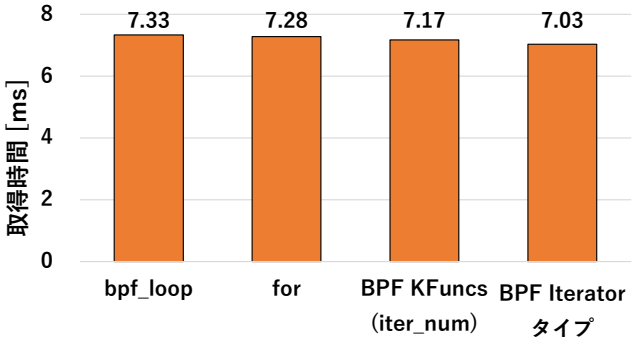

図 12 ループ方法の違いによるプロセス一覧の取得性能

直接メモリ参照と比べると 4.9 倍の取得時間となった．

5.4 ループ方法による性能差

eBPF プログラムで行うループ処理の方法には様々なトレードオフがある．そこで，ループ方法の監視性能への影響を調べるために，OS データをページ単位で先読みしてプロセス一覧の取得を行う時間を測定した．収集するページ数を揃えるために，情報を取得するプロセスを 150 個に統一した．プロセス一覧の取得時間の測定を 100 回行った時の平均値を図 12 に示す．BPF KFuncs の `iter_num` 関数と BPF Iterator タイプはループ処理を eBPF プログラム外で行い，BPF Iterator タイプはプロセス構造体の探索も eBPF プログラム外で行う．そのため，eBPF プログラム外で処理する部分が増えるほど，ループ処理が高速になると考えられる．しかし，ループ方法の違いによる性能差は取得時間全体と比べると小さいことが分かった．

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 [11] 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 [12] 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

© 1959 Information Processing Society of Japan

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 [13] 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 [14] 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 [15] を用いることができる．

7. まとめ

本稿では，Confidential 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，高速に必要な 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，複雑な OS データを一括取得する際のオーバヘッドを削減し，eBPF プログラムを手動で最適化した時の性能に近づけられるようにすることである．その上で，eBPFmonitor で一括取得できる OS データを増やす．また，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，本研究成果の一部は，国立研究開発法人情報通信研究機構（NICT）の委託研究（JPJ012368C05501）により得られたものである．

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 2024-11-3).
[4] LLVM Project: Writing an LLVM Pass, The LLVM Foundation (online), available from ⟨https://llvm.org/docs/WritingAnLLVMNewPMPass.html⟩ (accessed 2024-11-4).
[5] Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. Network and Distributed System Security Symposium* (2003).
[6] Ozaki, Y., Kanamoto, S. and Kourai, K.: Detection System Failures with GPUs and LLVM, *Proc. Asia Pacific Workshop on Systems*, pp. 47–53 (2019).
[7] IO Visor Project: BPF Compiler Collection (BCC), GitHub (online), available from ⟨https://github.com/iovisor/bcc⟩ (accessed 2024-11-3).
[8] The Linux Kernel: BPF Type Format (BTF), The Linux Kernel documentation (online), available from ⟨https://www.kernel.org/doc/html/latest/bpf/btf.html⟩ (accessed 2024-11-3).
[9] The Linux Kernel: BPF Iterators, The Linux Kernel documentation (online), available from ⟨https://docs.kernel.org/bpf/bpf_iterators.html⟩ (accessed 2024-11-3).
[10] The Linux Kernel: BPF Kernel Functions (kfuncs), The Linux Kernel documentation (online), available from ⟨https://docs.kernel.org/bpf/kfuncs.html⟩ (accessed 2024-11-3).
[11] 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).
[12] 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).
[13] Fernandes, J.: BPFd (Berkeley Packet Filter Daemon), GitHub (online), available from ⟨https://github.com/joelagnel/bpfd⟩ (accessed 2024-11-3).
[14] 堀恭介，光来健一：eBPF プログラムによる安全かつ透過的な VM 内のシステム監視機構，*CSEC* (2024).
[15] 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