AMD SEV-SNPのVMPLによるP4プログラムの軽量な隔離実行

村上 和馬[1,a)]　光来 健一[1]

概要：近年，P4 プログラムを実行可能な仮想 P4 スイッチが開発されている．仮想マシン (VM) のユーザが用意した P4 プログラムを使うことができれば柔軟なパケット転送処理が行えるが，信頼できないクラウドにおいては P4 プログラムが攻撃を受けるリスクがある．そこで，先行研究では保護された P4 VM 内で P4 プログラムを実行し，ユーザ VM 内の情報を安全に利用できるようにしている．しかし，ユーザごとに P4 VM を用意する必要があり，ユーザ VM から P4 VM に明示的に情報を渡す必要がある．本稿では，AMD SEV-SNP の VM 特権レベル (VMPL) を用いて軽量かつ安全に P4 プログラムを実行可能にする Parasite-P4 を提案する．P4 VM を用いず，ユーザ VM 内で P4 プログラムを高い権限で実行することにより，システムのメモリ上の情報を直接，取得することができる．P4 プログラムからシステムを保護するために，eBPF を用いて P4 プログラムを安全に実行する．Parasite-P4 を SVSM に実装し，パケットフィルタリングが行えることの確認，および性能の測定を行った．

VMPL-based Lightweight Isolated Execution of P4 Programs with AMD SEV-SNP

Kazuma Murakami[1,a)]　Kenichi Kourai[1]

Abstract: Recently, virtual P4 switches that can execute P4 programs have been developed. Flexible packet forwarding is possible if users can use P4 programs for each virtual machine (VM), but there is a risk that P4 programs may be attacked by untrusted clouds. Previous work enables P4 programs to run in protected P4 VMs and use information inside a user VM. However, it needs to prepare P4 VMs and explicitly pass information from user VMs to P4 VMs. This paper proposes Parasite-P4 for enabling the secure execution of P4 programs with VM privilege levels (VMPL) of AMD SEV-SNP. Parasite-P4 executes P4 programs with high privileges inside user VMs and directly obtains information from system memory. To protect the system, it safely executes P4 programs using eBPF. We have implemented Parasite-P4 in SVSM and examined its effectiveness and performance.

1. はじめに

近年，ネットワークスイッチにおいて P4 言語 [1] で書かれたプログラムを実行することにより，スイッチの機能を拡張してネットワークセキュリティを向上させることが可能になっている．仮想マシン（VM）が接続される仮想ネットワークについても P4 に対応した仮想 P4 スイッチ [2] が開発されており，パケット転送処理をプログラム可能になっている．VM の管理者が仮想 P4 スイッチに P4 プログラムをロードできるようになれば，VM ごとにパケット転送処理をカスタマイズしたり，VM 内のシステム情報を利用したりすることもできるようになる．しかし，信頼できないクラウドにおいては，ユーザが仮想 P4 スイッチにロードした P4 プログラムが改ざんされたり，P4 プログラムが取得した VM 内のシステム情報が盗聴されたりする可能性がある．逆に，不正な P4 プログラムが仮想 P4 スイッチにロードされた場合，仮想スイッチ全体に影響を及ぼし，正常なパケット転送が行われなくなる可能性がある．

そこで，メモリ等が保護された機密 VM をユーザごと

[1] 九州工業大学
Kyushu Institute of Technology
[a)] murakami@ksl.ci.kyutech.ac.jp

に用意し，その中で安全に P4 プログラムを実行する P4 Shield [3] が提案されている．仮想スイッチは用意した P4 VM にパケットを転送し，仮想スイッチから隔離された環境で安全に P4 プログラムを実行する．その実行結果に基づいて仮想スイッチはパケットの転送処理を行う．また，ユーザ VM が P4 VM とシステム情報を共有することで，P4 プログラムがユーザ VM 内の情報を用いることを可能にしている．しかし，P4 Shield ではユーザごとに P4 VM を用意する必要があるため，その分のシステムリソースが余分に必要になり，クラウドやユーザのコスト上昇につながる．それに加えて，P4 プログラムはユーザ VM が明示的に共有したシステム情報しか参照できないため，P4 プログラムが新たな情報を必要とした際にはユーザ VM 内のシステムも合わせて拡張する必要がある．

本稿では，AMD SEV-SNP によって提供される VM 特権レベル（VMPL） [4] を用いてユーザ VM 内で P4 プログラムを安全に実行することを可能にする Parasite-P4 を提案する．ユーザ VM を活用することにより，ユーザごとに P4 VM を用意するよりも少ないシステムリソースで P4 プログラムを実行することができる．Parasite-P4 は P4 プログラムを同じユーザ VM 内で動作するシステムから保護するために，P4 プログラムを高い特権レベルで動作させ，システムを低い特権レベルで動作させる．さらに，高い特権レベルからは低い特権レベルのメモリにアクセスすることができるため，P4 プログラムはシステムのメモリ上の情報を柔軟に取得することができる．逆に，P4 プログラムがシステムに影響を及ぼすのを防ぐために，eBPF [5] を用いて P4 プログラムを安全に実行する．

Parasite-P4 を COCONUT Secure VM Service Module (SVSM) [6] と Linux カーネルを用いて実装した．P4 プログラムは eBPF バイトコードにコンパイルし，メモリ安全な Rust 言語を用いて実装された SVSM 上で動作する rbpf ランタイム [7] を用いて実行する．eBPF バイトコードから共有メモリやシステムメモリにアクセスできるようにするために，rbpf ランタイムがヘルパー関数を提供する．このヘルパー関数はアドレス変換やメモリマッピングを行い，これらの外部メモリにアクセスして eBPF バイトコードにデータを返す．実験の結果，ユーザ VM 内の P4 プログラムがシステム情報を用いてパケットの転送・破棄を判断できることを確認した．また，P4 プログラムの実行にかかる時間を測定し，P4 VM を用いる P4 Shield との比較を行った．

以下，2 章で仮想 P4 スイッチにユーザの P4 プログラムをロードする際の問題点と先行研究の P4 Shield について述べる．3 章で VMPL を用いてユーザ VM 内で P4 プログラムを安全に実行する Parasite-P4 を提案し，4 章でその実装について説明する．5 章で Parasite-P4 の性能や使用リソース量を確かめるために行った実験について述べる．6 章で関連研究を示し，7 章で本稿をまとめる．

2. クラウドにおける仮想 P4 スイッチ

P4 言語 [1] を用いてデータプレーンをプログラム可能なネットワークスイッチが用いられるようになっている．従来，新しいパケットフィルタリング手法や侵入検知システムを利用したい時にはそれに対応したスイッチの開発を待つ必要があった．P4 スイッチを用いればスイッチ内で P4 プログラムを実行することで新しい機能に対応させることが可能である．仮想ネットワークにおいても P4 スイッチと同等の機能を提供する仮想 P4 スイッチ [2] が開発されている．仮想ネットワークは VM の仮想 NIC を同一ホスト上にある仮想スイッチに接続することで構築され，さらに仮想スイッチを物理 NIC に接続することで外部のネットワークとも接続される．そのため，仮想 P4 スイッチが転送するすべてのパケットに P4 プログラムを適用することができる．

物理 P4 スイッチにはネットワーク管理者が P4 プログラムをロードするが，仮想 P4 スイッチには仮想ネットワークまたはホストの管理者が P4 プログラムをロードすることになる．さらに，VM の管理者が仮想 P4 スイッチに P4 プログラムをロードできるようになれば，VM ごとにカスタマイズされたパケット転送処理を行うことが可能になる．これにより，例えば，DoS 攻撃を受けた VM の管理者が即座に P4 プログラムをロードし，VM にパケットが届く前に通信を遮断することができるようになる．また，P4 プログラムがパケット情報だけでなくパケットを送受信する VM 内のシステム情報も用いることができれば，よりきめ細かいパケット処理を行うことが可能になる．例として，パケットを送受信するプロセスやユーザの情報を利用したパケットフィルタリング [8] が挙げられる．

しかし，クラウドにおいて VM の管理者（ユーザ）の P4 プログラムを仮想 P4 スイッチで実行できるようにするといくつかの問題が生じる．第一に，クラウドが提供している仮想 P4 スイッチはユーザにとって信頼できるとは限らない．悪意のあるクラウド管理者などの内部犯が存在する場合，ユーザがロードした P4 プログラムを改ざんされる恐れがある．また，P4 プログラムを盗み見られたり，P4 プログラムが取得した VM 内のシステム情報を盗聴される可能性もある．第二に，ユーザの P4 プログラムはクラウドにとって信頼できるとは限らない．P4 プログラムが計算リソースを占有してしまい，仮想 P4 スイッチ全体に影響を与える可能性がある．実際に，P4 プログラムやそれを実行するランタイムの脆弱性により，様々な攻撃が可能であることが指摘されている [10]．

そこで，先行研究の P4 Shield [3] では P4 プログラムを仮想スイッチ上ではなく，メモリ等が保護された機密 VM 内で実行することにより，P4 プログラムと仮想スイッチを

相互に保護している．仮想スイッチはパケットを受信するとそれをユーザごとに用意された P4 VM に転送し，隔離された環境で P4 プログラムを安全に実行する．クラウドは機密 VM 内の P4 プログラムを攻撃することはできず，P4 プログラムは P4 VM の外で動作する仮想スイッチに影響を及ぼすことはできない．P4 VM 内で実行される複数の P4 プログラムを互いに保護するために，P4 プログラムは uBPF [11] を用いて独立した実行環境で実行する．P4 プログラムがユーザ VM 内のシステム情報を利用することができるようにするために，ユーザ VM が明示的に P4 VM に対して情報を共有する．

P4 Shield では，P4 VM を用いることでユーザの P4 プログラムの安全な実行を実現しているが，ユーザ間の攻撃を防ぐためにユーザごとに P4 VM を用意している．P4 VM を動作させるには，CPU やメモリ，ディスクなどのシステムリソースが必要になる．ユーザが P4 VM のためのコストを負担する場合，ユーザのコスト増となる．一方で，クラウドが P4 VM のコストを負担する場合，クラウドのコスト増となり，結果的にユーザ VM の利用料金に反映されることになる．また，P4 プログラム内で VM のシステム情報を用いるには，P4 プログラムが必要とする情報をユーザ VM があらかじめ共有しておく必要がある．例えば，通信に関する詳細な情報が必要な場合，その情報を共有メモリに格納しておくことになる．必要とする情報は P4 プログラムごとに異なるため，新たなシステム情報を利用する場合にはユーザ VM 内のシステムを拡張することが必要になる．

3. Parasite-P4

本稿では，AMD SEV-SNP によって提供される VM 特権レベル（VMPL） [4] を用いてユーザ VM 内で P4 プログラムを安全に実行することを可能にする Parasite-P4 を提案する．SEV-SNP は機密 VM を作成するために用いられる AMD プロセッサのセキュリティ機構であり，VMPL は機密 VM 内のメモリを 4 つの特権レベルに分割することができる機能である．Parasite-P4 は**図 1** のようにユーザ VM 内で P4 プログラムを最も高い特権レベルである VMPL0 で動作させ，システムをより低い特権レベルで動作させる．仮想スイッチはパケットを受信すると共有メモリ経由でユーザ VM にパケットを転送し，ユーザ VM の特権レベルを VMPL0 に切り替えることによって P4 プログラムを実行する．VMPL を用いることにより，P4 Shield のように P4 VM を別途，用意することなく，P4 プログラムを隔離して実行することができる．その結果，P4 VM を用いる場合よりも少ないシステムリソースで P4 プログラムを実行することができる．

Parasite-P4 は P4 プログラムを既存のユーザ VM 内で動作させるが，VMPL を用いることにより，同じユーザ VM

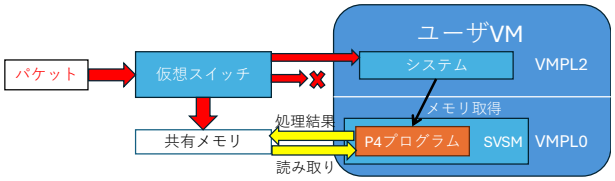

図 1 Parasite-P4 のシステム構成

内で動作するゲスト OS が攻撃を受けた場合であっても P4 プログラムを保護することができる．また，SEV-SNP によるユーザ VM の保護により，P4 VM を用いる場合と同様に，仮想スイッチからユーザ VM 内で動作する P4 プログラムの盗聴や改ざんを行うことはできない．P4 プログラムはユーザ VM 内で実行されるため，P4 VM 内で実行する場合と同様に，VM 外部の仮想スイッチを攻撃することもできない．その一方で，VMPL0 からはより低い特権レベルで動作するシステムのメモリにアクセスすることができるため，P4 プログラムが必要とするシステム情報を直接，取得することができる．システム情報を提供するためにユーザ VM 内のシステムを拡張する必要がないため，ユーザ VM がシステム情報を明示的に共有メモリに格納する P4 Shield よりも柔軟性が高い．

一方，VMPL0 で動作する P4 プログラムは同じユーザ VM 内で動作するシステムに影響を及ぼす可能性があるため，Parasite-P4 は eBPF [5] を用いて P4 プログラムを実行する．eBPF は OS カーネル内で安全にプログラムを実行してシステムを監視したりするために用いられているフレームワークである．Parasite-P4 は P4 プログラムを eBPF バイトコードにコンパイルし，検証器を用いて安全性を検査することにより，P4 プログラムを安全に実行する．P4 VM のユーザ空間で P4 プログラムを実行する P4 Shield とは異なり，VMPL0 で eBPF バイトコードを実行するランタイムに脆弱性があると P4 プログラムに VMPL0 の特権を取得されてしまう危険性がある．そこで，Parasite-P4 はメモリ安全なプログラミング言語である Rust で記述されたランタイムを用いることでメモリ関連の脆弱性を削減し，セキュリティを向上させる．

Parasite-P4 はユーザ VM 内のシステム情報も eBPF バイトコードを用いて安全かつ柔軟に取得することができる．P4 Shield では，ユーザ VM 内のゲスト OS によって共有メモリに格納されたシステム情報を取得する機能が P4 プログラムに提供されているだけであった．それに対して，Parasite-P4 ではシステム情報を取得するためにシステムのメモリ上にある OS データを解析するプログラムを P4 プログラムから呼び出して実行することができる．OS データの解析は OS のソースコードを利用して記述し，eBPF バイトコードにコンパイルする際に VMPL0 からゲスト OS のメモリにアクセスするコードを埋め込む．これにより，P4 プログラムの eBPF バイトコードと解析プログラ

ムの eBPF バイトコードを統合することが可能になる．

ただし，セキュリティ上，eBPF バイトコードから直接，ゲスト OS のメモリや共有メモリにアクセスすることはできないため，これらの外部メモリにアクセスするためのヘルパー関数を eBPF ランタイムが提供する．eBPF バイトコードはヘルパー関数を呼び出すことにより，ゲスト OS のメモリの読み込み，および共有メモリの読み書きを行う．ヘルパー関数がゲスト OS のメモリにアクセスする際には，ゲスト OS のページテーブルを参照して OS データの仮想アドレスを物理アドレスに変換し，ゲスト OS のメモリを VMPL0 にマッピングすることにより OS データを取得する．共有メモリにアクセスする際には，共有メモリを VMPL0 にマッピングすることにより，パケットデータの読み込みや P4 プログラムの実行結果の書き込みを行う．

4. 実装

Parasite-P4 を COCONUT Secure VM Service Module (SVSM) [6] と Linux カーネルに実装した．

4.1 SVSM による OS データの取得

SVSM は SEV-SNP の VMPL0 で動作するソフトウェアであり，SEV-SNP を用いて保護された機密 VM 内で動作するゲスト OS に対して安全なサービスやデバイスエミュレーションを提供する．SVSM は主に Rust で書かれており，高い特権レベルで動作させても安全性が高い．SVSM は SVSM バイナリと UEFI の VM 向け実装である OVMF を一つにまとめた IGVM 形式のファイルを用いて起動する．VM を起動するとまず，VMPL0 で SVSM を起動して初期化を行う．その後，低い特権レベルである VMPL2 に切り替えて OVMF を起動し，その上でゲスト OS の Linux を起動する．ゲスト OS の起動後は，VMPL0 に切り替えることによって SVSM を実行する．

SVSM はゲスト OS のもつシステム情報を取得するために，ゲスト OS のページテーブルを用いて OS データの仮想アドレスを物理アドレスに変換する．ゲスト OS の実行時にはページテーブルの物理アドレスが CR3 レジスタに格納されている．そのレジスタの値は VMPL0 に切り替わる際に CPU によって VMSA と呼ばれる領域に保存される．SVSM は**図 2** のように VMSA から CR3 レジスタの値を取得し，ゲスト OS のページテーブルを特定する．VMSA は暗号化されているため VM 外部からは参照することができないが，VM 内部の VMPL0 で動作する SVSM は参照することができる．SVSM はゲスト OS のページテーブルエントリを自身の仮想メモリにマッピングすることにより，ページテーブルをたどってアドレス変換を行う．そして，最終的に得られた変換先の物理ページをマッピングすることにより，OS データにアクセスする．

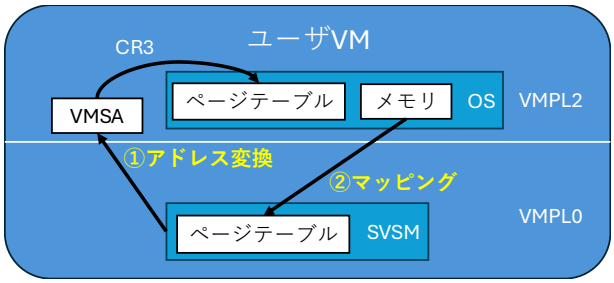

図 2 OS データの取得の流れ

4.2 SVSM での共有メモリの利用

仮想スイッチとユーザ VM 間でパケットデータや P4 プログラムの実行結果をやりとりするために，QEMU の ivshmem [12] を用いて共有メモリを確立する．そのために，ホスト OS 上で ivshmem-server を実行し，共有メモリとして用いるメモリ領域を確保してユーザ VM に提供する．ユーザ VM にはそのメモリ領域にアクセスするための仮想 PCI デバイスが作成される．ユーザ VM 内では，この仮想 PCI デバイスを扱うための ivshmem-uio ドライバ [13] を Linux カーネルにロードする．Linux カーネル内では仮想 PCI デバイスから共有メモリの物理アドレスを取得し，共有メモリをマッピングすることで直接，アクセスすることができる．この際に，SEV-SNP によるメモリ暗号化が行われないように設定されるため，VM 外部の仮想スイッチとの間でデータのやりとりが可能である．この共有メモリに対応するデバイスファイルがホスト OS 上に作成されるため，仮想スイッチはデバイスファイルをメモリにマッピングすることで直接，共有メモリの読み書きを行うことができる．

SVSM 用の ivshmem-uio ドライバは存在しないため，Parasite-P4 は共有メモリの物理アドレスをゲスト OS から SVSM に登録することで SVSM から共有メモリへのアクセスを可能にする．共有メモリの物理アドレスは ivshmem-uio ドライバが SVSM コールを用いて SVSM を呼び出すことによって登録する．SVSM コールは VMGEXIT 命令を実行してハイパーバイザを呼び出し，ハイパーバイザが VMPL2 から VMPL0 に切り替えることで実現されている．SVSM コールには様々なプロトコルが定義されており，共有メモリの物理アドレスを登録するために追加したプロトコルを指定する．VMGEXIT 命令は様々な用途に用いられるため，Guest-Hypervisor Communication Block (GHCB) と呼ばれるメモリ領域に SVSM コールであることを表す情報を書き込んでから VMGEXIT 命令を実行する．GHCB はレジスタの状態が暗号化される SEV-SNP において，ハイパーバイザと情報を共有するために用いられる．これらの SVSM コールの処理の様子を**図 3** に示す．

SVSM コールを用いて登録された共有メモリの物理アドレスを用いて，SVSM は共有メモリをマッピングし，共有メモリ上のデータにアクセスする．そのために，SVSM の

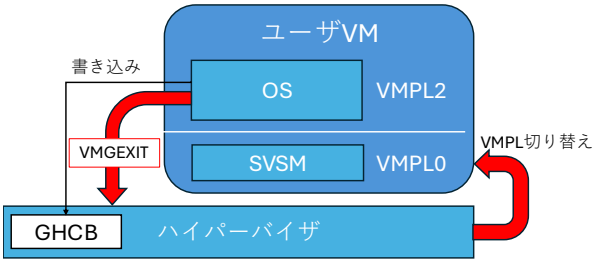

図 3 SVSM コールの処理の流れ

ページテーブルに共有メモリの物理アドレスを登録する．その際に，SEV-SNP によって暗号化されてしまうと仮想スイッチとのやりとりが行えなくなるため，暗号化されないようにページテーブルを設定する．具体的には，ページテーブルエントリの C ビットで暗号化の有無を決定できるため，C ビットをクリアすることで暗号化されないようにする．現在の実装では，共有メモリのマッピングを SVSM コール後に保持することができていないため，SVSM が共有メモリにアクセスする時に毎回マッピングを行っている．

4.3 SVSM との通信

仮想スイッチは共有メモリにパケットデータを書き込んだ後，ユーザ VM 内の SVSM を呼び出して P4 プログラムを実行させる必要があるが，SVSM には外部と直接通信する機構は用意されていない．そこで，ユーザ VM のゲスト OS 上でプロキシを動作させ，プロキシ経由で SVSM を呼び出す手法が考えられる．この手法では，仮想スイッチが仮想ネットワーク経由でユーザ VM 内のプロキシと通信し，プロキシが Linux カーネルとハイパーバイザ経由で SVSM を呼び出す．この手法は仮想ネットワークを用いた通信を必要とするため，仮想スイッチからユーザ VM 内のプロキシに向けて送出されたパケットは転送処理を行うために再び，仮想スイッチに送られる．そのパケットに対して SVSM を呼び出すためにプロキシと通信することになるため，仮想スイッチで特殊なパケット処理を行わない限り，プロキシとの通信を無限に繰り返すことになる．

そこで，ユーザ VM 内で共有メモリをポーリングすることによってパケットデータの書き込みを検知することが考えられるが，SVSM 内ではポーリングを行うことができない．ポーリングによって SVSM が仮想 CPU を占有してしまうと，ユーザ VM 内のシステムがその仮想 CPU を使用できなくなるためである．そこで，Parasite-P4 はゲスト OS 内で共有メモリのポーリングを行い，パケットデータの書き込みを検知すると SVSM を呼び出す．ポーリングによって仮想 CPU が占有されるの防ぐために，Linux の高精度カーネルタイマを用いて $10\mu s$ おきに共有メモリのチェックを行う．SVSM を呼び出す際には，P4 プログラムを実行するために追加したプロトコルを指定して SVSM コールを実行する．SVSM が P4 プログラムを実行している間はタイマを停止してポーリングが行われないようにし，SVSM コールが終了すると再び，タイマを設定してポーリングを再開する．

4.4 rbpf ランタイムの拡張

Parasite-P4 は SVSM に組み込んだ rbpf ランタイム [7] を用いて，eBPF バイトコードにコンパイルされた P4 プログラムを実行する．rbpf は eBPF のユーザ空間実装である uBPF [11] をベースに Rust で実装されたランタイムである．SVSM も Rust で実装されているため，rbpf ランタイムは SVSM から容易に呼び出すことができる．rbpf ランタイムは OS 上のユーザ空間での実行を前提としているが，SVSM のように OS の標準ライブラリが提供されていない環境であっても no_std 環境を用いることで動作させることが可能である．ただし，no_std 環境では eBPF バイトコードの JIT コンパイルが行えなかったり，デバッグ機能が制限されたりする．

Parasite-P4 は SVSM のメモリ管理機構を利用することで，rbpf ランタイムが eBPF バイトコードの JIT コンパイルを行うことを可能にする．JIT コンパイルには読み書きと実行のすべてが許可されたメモリ領域が必要となる．標準ライブラリでは `mprotect` 関数を用いてメモリ保護を変更することができるが，SVSM ではこのような機能は提供されていない．そこで，SVSM の機能を用いて通常のメモリ確保を行った後，SVSM のページテーブルをたどってメモリの仮想アドレスを物理アドレスに変換する．そして，読み書き可能かつ実行可能になるようにページテーブルに設定を行って再マッピングする．このメモリを JIT コンパイル用のメモリとして rbpf ランタイムに登録し，JIT コンパイルしたコードを配置して実行できるようにする．

eBPF バイトコードは割り当てられたメモリ以外にはアクセスできないため，Parasite-P4 は外部メモリにアクセスするためのヘルパー関数を提供する．eBPF バイトコードはヘルパー関数に割り当てられた番号をアドレスとして `call` 命令を実行することにより，rbpf ランタイムによって提供されているヘルパー関数を実行することができる．現在のところ，ゲスト OS のカーネルの仮想アドレスを指定して 64 ビットの OS データを取得するヘルパー関数と，共有メモリの物理アドレスを指定して 64 ビットのデータを読み書きするヘルパー関数が利用可能である．これらの関数を rbpf ランタイムに登録し，eBPF バイトコードから呼び出せるようにする．

5. 実験

Parasite-P4 がユーザ VM 内の VMPL0 で P4 プログラムと OS データの解析プログラムを実行し，パケットフィルタリングの判断を行えることを確認する実験を行った．また，仮想スイッチがパケットをユーザ VM に転送して，

表 1　実験環境

	ホスト	ユーザ VM	P4 VM
CPU	AMD EPYC 7713P (64 コア)	8 コア	8 コア
メモリ	128GB	8GB	8GB
OS	Linux 6.11	Linux 6.11	Linux 6.8
ハイパーバイザ	QEMU-KVM 9.1.50	–	–

P4 プログラムの実行結果を受け取るまでの時間を計測した．比較として，P4 Shield においてパケットを P4 VM に転送してから実行結果を受け取るまでの時間も計測した．Open vSwitch などの仮想スイッチにはまだ実装できていないため，仮想スイッチを模した簡易プログラムを用いた．また，P4 プログラムと解析プログラムは C 言語で実装して，eBPF バイトコードにコンパイルしたものを用いた．実験環境を**表 1** に示す．OS とハイパーバイザには SEV-SNP と SVSM に対応済みのもの [6] を用いた．

5.1　動作確認

Parasite-P4 を用いて，ユーザ VM 内の P4 プログラムがゲスト OS の状態を用いてパケットの転送・破棄を判断できるかどうかを調べた．ゲスト OS の状態として，Linux カーネル内の `tcp_memory_allocated` 変数の値を取得した．このグローバル変数には TCP が使用しているメモリページ数が格納されている．ユーザ VM を起動した直後は 0 であり，TCP 通信を行うと増加する．この実験では，この変数の値が小さい時，つまり，ゲスト OS 上で TCP 通信があまり行われていない時にのみパケット転送を許可するようにした．実際に，Linux カーネルではこの変数の値がしきい値を超えた場合に送信制御を行っている．

まず，ユーザ VM の起動直後に，仮想スイッチから共有メモリにダミーのパケットデータを書き込んだ．その結果，ユーザ VM 内で P4 プログラムと OS データの解析プログラムが実行され，転送を許可するメッセージが共有メモリに書き込まれたことが仮想スイッチにおいて確認できた．次に，ユーザ VM と他の VM との間で iperf を用いて TCP 通信を行っている間に，仮想スイッチから共有メモリにパケットデータを書き込んだ．この時，ゲスト OS において `tcp_memory_allocated` 変数の値が大きくなり，P4 プログラムによってパケットを破棄するメッセージが共有メモリに書き込まれたことが確認できた．この結果より，Parasite-P4 を用いてパケットの転送・破棄が正しく判断できることがわかった．

5.2　P4 プログラムの実行時間

Parasite-P4 を用いて，仮想スイッチが共有メモリにパケットデータを書き込んでから，P4 プログラムによって共有メモリに書き込まれた実行結果を受け取るまでにかかる時間を測定した．この P4 プログラムはヘルパー関数を

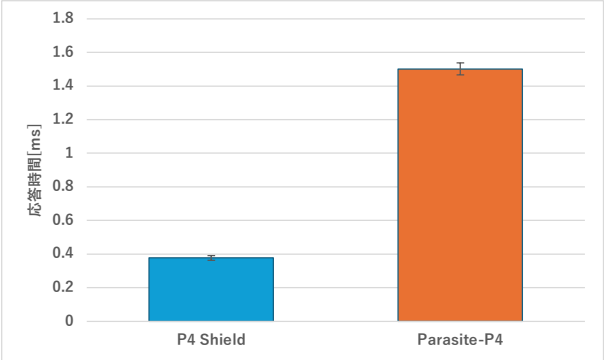

図 4　P4 プログラムの実行時間

3 回呼び出して，共有メモリに格納されたパケットデータの取得，5.1 節の OS データの取得，転送・破棄の判断の共有メモリへの書き込みを行った．これを P4 Shield と比較するために，仮想スイッチが仮想ネットワークを用いて P4 VM にパケットデータを送信してから，P4 プログラムの実行結果を受信するまでの時間も測定した．この P4 プログラムはゲスト OS のメモリを解析して OS データを取得する代わりに，ユーザ VM との間の共有メモリにあらかじめ書き込んでおいたゲスト OS の状態を取得した．

それぞれ 10 回ずつ P4 プログラムの実行時間を測定した結果を**図 4** に示す．Parasite-P4 では平均 1.50ms，P4 Shield では平均 0.38ms となり，P4 Shield の方が大幅に短いことがわかった．P4 Shield では P4 VM 内でパケットデータを受信するとすぐに P4 プログラムを実行できるのに対し，Parasite-P4 では VMPL を切り替えてから P4 プログラムを実行する必要がある．また，P4 Shield ではパケットデータは P4 プログラムから直接アクセスすることができるため，ヘルパー関数を用いて読み書きを行う必要がない．その上，ゲスト OS の状態が書き込まれた共有メモリは事前にマッピングしているため，P4 プログラムの実行中にマッピング処理を行う必要がない．一方，Parasite-P4 ではパケットデータを読み書きするヘルパー関数を呼び出すたびにメモリのマッピングを行う必要がある．加えて，ゲスト OS のメモリにアクセスするヘルパー関数を呼び出すたびに，ゲスト OS の仮想アドレスを物理アドレスに変換してメモリのマッピングを行う必要がある．これらが Parasite-P4 において P4 プログラムの実行時間が遅い原因である．SVSM においてメモリのマッピングを保持するように実装することができれば，オーバヘッドを削減できると考えられる．

6. 関連研究

P4rt-OVS [2] は Open vSwitch [14] ベースの仮想 P4 スイッチである．物理スイッチとは異なり，仮想スイッチはソースコードを変更することで新しい機能に対応させることも可能であるが，P4 プログラムを用いることで仮想スイッチの拡張を容易にしている．P4 プログラムは p4c-ubpf コンパイラ [15] を用いて eBPF バイトコードにコンパイルされ，ユーザ空間で動作する仮想 P4 スイッチ内で uBPF ランタイム [11] を用いて実行される．uBPF によって仮想スイッチは P4 プログラムから保護されるが，P4 プログラムは仮想スイッチからは保護されない．そのため，仮想スイッチが信頼できない環境では，ユーザの P4 プログラムをロードして安全に実行することはできない．また，uBPF ランタイムは C 言語で実装されているため，その脆弱性を P4 プログラムに悪用されると仮想スイッチに影響が及ぶ可能性がある．加えて，VM 内のシステム情報を P4 プログラムから利用することもできない．

P4 Shield [3] は保護された P4 VM を用いて，仮想スイッチから P4 プログラムを隔離して安全に実行する．P4 VM 内で実行される複数の P4 プログラムを互いに保護するために，それぞれの P4 プログラムは uBPF を用いて隔離して実行する．uBPF ランタイムに脆弱性があった場合は P4 プログラムが他の P4 プログラムを攻撃できる可能性があるが，その場合でも仮想スイッチには影響が及ばない．P4 VM は多くのシステムリソースを必要とするが，Parasite-P4 のように VMPL でユーザ VM のメモリを分割するよりも強いリソース分離を行うことができる．そのため，P4 プログラムがリソースを占有した場合には VM 単位でリソース制限を行うことにより，仮想スイッチやユーザ VM への影響を抑えることができる．一方，Parasite-P4 の場合は P4 プログラムによるリソース占有の影響が同じユーザ VM 内のシステムに及ぶ可能性がある．また，P4 プログラムを実行するために VMPL を切り替える必要がないため，より高速に動作する．

EndBox [16] は Intel SGX を用いてファイアウォールや侵入検知システムなどのミドルボックス機能をクライアント側で安全に提供する．SGX はエンクレイヴと呼ばれる保護領域をユーザ空間に作成し，メモリの暗号化や整合性検査によってエンクレイヴ内のプログラムを安全に実行することを可能にする．EndBox はミドルボックスをエンクレイヴ内で実行し，エンクレイヴ内に VPN エンドポイントを配置することですべてのトラフィックに対してミドルボックスの通過を強制している．SGX-Box [18] はエンクレイヴ内で動作するミドルボックスが暗号化されたパケットを安全に復号して侵入検知などを行うことを可能にしている．そのために，高水準言語である SB lang を提供しており，パケット処理や暗号処理の詳細を隠蔽することで，ミドルボックスのプログラミングを容易にしている．

エンクレイヴは少ないシステムリソースで動作するため，仮想スイッチ内にエンクレイヴを作成して P4 プログラムを実行することも考えられる．しかし，P4 プログラムと仮想スイッチ間のリソース分離は VMPL を用いる場合よりも弱くなるため，P4 プログラムがリソースを占有すると仮想スイッチにも影響が及ぶ．VMPL を用いる場合のようにユーザ VM に影響が及ぶことはないが，ユーザ VM 間で共有される仮想スイッチへの影響の方がシステム全体への影響が大きい．また，エンクレイヴは Intel SGX や RISC-V Keystone [17] などでのみサポートされており，AMD を含めて多くのプロセッサでサポートされている機密 VM を用いるよりも汎用性が低い．

00SEVen [19] は AMD SEV-SNP で提供される VMPL を用いることで，機密 VM の保護を弱めることなくリモートからの VM イントロスペクション（VMI）を実現する．リクエストに応じて，高い特権レベルで動作する VMI エージェントがゲスト OS のメモリからデータを取得してリモートホストに返送する．VMI エージェントは VSOCK [20] を用いてホスト OS 上のプロキシ経由でリモートホストと通信する．プロキシがリモートホストからパケットを受信した際には，直接，ハイパーバイザを呼び出して VMPL を切り替える．Parasite-P4 でも同様に，ホスト OS 上で動作する仮想スイッチがハイパーバイザを呼び出して VMPL を切り替えることも可能である．

7. まとめ

本稿では，AMD SEV-SNP の VMPL によって隔離された実行環境を用いることで，ユーザの VM 内で P4 プログラムを軽量かつ安全に実行する Parasite-P4 を提案した．Parasite-P4 はクラウドやユーザ VM 内のシステムへの侵入者から P4 プログラムを保護すると同時に，不正な P4 プログラムから仮想スイッチを保護する．さらに，eBPF を用いることで P4 プログラムからユーザ VM 内のシステムも保護する．Parasite-P4 を SVSM と Linux カーネルに実装し，ユーザ VM 内のシステムの状態を参照する P4 プログラムを用いて実験を行った．実験結果より，ゲスト OS のメモリから取得した状態を用いてパケットの転送・破棄の判断が行えることを確認した．また，Parasite-P4 は P4 Shield よりもオーバヘッドが大きいこともわかった．

今後の課題は，P4 プログラムの実行性能を改善することである．その上で，Open vSwitch などの仮想スイッチを用いて実装を行い，Parasite-P4 のネットワーク性能への影響を詳細に調べる．また，P4 言語で書かれた P4 プログラムと C 言語で書かれた OS データの解析プログラムを統合できるようにすることも必要である．

謝辞 本研究の一部は，JST 経済安全保障重要技術育成プログラム【JPMJKP24U4】の支援を受けたものである．

参考文献

[1] P4 Project: P4 Open Source Programming Language, https://p4.org/.

[2] Tomasz Osiński, Halina Tarasiuk, Paul Chaignon, Mateusz Kossakowski: P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch with P4, In *Proceedings of IFIP Networking Conference*, pp. 413–421, 2020.

[3] 岩井 正輝，光来 健一: VM 内情報を利用する仮想 P4 スイッチの安全な実行, セキュリティサマーサミット 2024, 2024.

[4] Advanced Micro Devices, Inc.: AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, White Paper, 2020.

[5] A. Starovoitov and D. Borkmann: eBPF – Introduction, Tutorials & Community Resources, https://ebpf.io/, 2014.

[6] SUSE: COCONUT Secure VM Service Module (SVSM), https://github.com/coconut-svsm/svsm.

[7] Quentin Monnet: rbpf: Rust Virtual Machine and JIT compiler for eBPF Programs, https://github.com/qmonnet/rbpf.

[8] Abhinav Srivastava, Jonathon Gifin: Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors, *Recent Advances in Intrusion Detection*, pp. 39–58, Berlin Heidelberg, 2008. Springer Berlin Heidelberg.

[9] Tomasz Osiński: P4-OvS: Bringing the power of P4 to OvS!, https://github.com/osinstom/P4-OvS.

[10] Mihai Valentin Dumitru, Dragos Dumitrescu, Costin Raiciu: Can We Exploit Buggy P4 Programs?, In *Proceedings of the Symposium on SDN Research*, pp. 62–68, 2020.

[11] IO Visor Project: uBPF: Userspace eBPF VM, https://github.com/iovisor/ubpf.

[12] QEMU Project Developers: Inter-VM Shared Memory Device, https://www.qemu.org/docs/master/system/devices/ivshmem.html.

[13] Shawn Anastaio: A Fork of Cam Macdonell's ivshmem Driver for Modern Kernels, https://github.com/shawnanastasio/ivshmem-uio.

[14] Linux Foundation: Open vSwitch. https://www.openvswitch.org.

[15] P4 Project: P4C, https://github.com/p4lang/p4c.

[16] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, Peter Pietzuch, Rüdiger Kapitza: EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution, In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 386–397, 2018.

[17] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song: Keystone: An Open Framefork for Architecting Trusted Execution Environments, In *Proceedings of the Fifteenth European Conference on Computer Systems* pp. 1–16, 2020.

[18] Juhyeng Han, Seongmin Kim Jaehyeong Ha, Dongsu Han: SGX-Box: Enabling Visibility on Encrypted Traffic using a Secure Middlebox Module, In *Proceedings of the First Asia-Pacific Workshop on Networking*, pp. 99–105, 2017.

[19] Fabian Schwarz and Christian Rossow: 00SEVen – Reenabling Virtual Machine Forensics: Introspecting Confidential VMs Using Privileged in-VM Agents, In *Proceedings of the 33rd USENIX Security Symposium*, pp. 1651–1668, 2024.

[20] Stefan Hajnoczi: virtio-vsock: Zero-configuration Host/guest Communication, KVM Forum 2015, 2015.