Vol.2019-OS-146 No.2
2019/5/30

XDP の実用化に向けた eBPF におけるユーザビリティの改善とその検討

小町 芳樹[1,a)]　牧田 俊明[1,b)]　小西 隆介[1,c)]　寺本 純司[1,d)]

概要：近年では，Linux を用いて NFV 環境を構築する場合などに高性能なネットワーク機能を要求されるが，Linux カーネルのネットワークスタックは高度に複雑化しているためオーバヘッドが大きく，性能面での要求を満たすことが難しい．こういった仮想ネットワークの性能を柔軟に改善する技術として，現在では XDP が注目を集めている．XDP は，Linux カーネルが提供する多様な機能を活かしつつ，ユーザが実装したプログラムを安全にカーネルに挿入できる eBPF を利用して，高速なネットワーク機能を実装できる技術だが，eBPF で挿入されるプログラムは実装面での制約が厳しいため，実用化の妨げになっている．そこで本研究では，難易度の高いネットワーク機能の実装を隠蔽する方式について検討を行った．本稿では一例として，eBPF による機能の挿入を既存の API を通じて透過的にライブラリもしくはカーネル内部で行うことで，従来の操作との互換性を保ちつつ，高速な転送機能を備えた L2 スイッチが実現できることを確認した．さらに隠蔽化の検討を通して，現在の eBPF に不足する機能や課題をユーザビリティの観点で整理し，eBPF そのものに対しても解決すべき課題を明らかにする．

キーワード：Linux ネットワーク，NFV，eBPF，XDP，ユーザビリティ

A Study on Improvement of eBPF Usability toward Practical XDP

Yoshiki Komachi[1,a)]　Toshiaki Makita[1,b)]　Ryusuke Konishi[1,c)]　Junji Teramoto[1,d)]

Abstract: In recent years, high-performance networking on Linux is demanded in environments such as NFV. However, it is hard to fulfill the performance requirements since highly complicated Linux networking stack has considerable amount of overhead. As a technology to resolve the performance problem in such virtual networks without losing flexibility, XDP is recently getting a lot of attention. On the basis of eBPF which makes it possible to safely insert user-implemented programs into Linux kernel, it enables us to implement high-performance network functions along with taking advantage of various features provided by the kernel. But, practical use of XDP is discouraged because eBPF has strict restriction in manner of its programming. We therefore studied methods to conceal the difficulties in programming network functions. First of all, this paper proposes a method to transparently insert eBPF-based features in a library or kernel through existing APIs. As an example, we design and implement a high-performance layer 2 switch compatible with existing operations in order to confirm the feasibility of the method. Furthermore, our examination results on the concealing method also clarified insufficient features and issues in usability of eBPF itself.

Keywords: Linux networking, NFV, eBPF, XDP, Usability

[1] NTT Open Source Software Center
1–9–1, Kounan, Minato-ku, Tokyo, 108–8019 Japan
[a)] yoshiki.komachi.vh@hco.ntt.co.jp
[b)] toshiaki.makita.vh@hco.ntt.co.jp
[c)] ryusuke.konishi.zy@hco.ntt.co.jp
[d)] junji.teramoto.hg@hco.ntt.co.jp

1. はじめに

1.1 背景

近年では，汎用のサーバを用いて柔軟にネットワーク機能を拡張することで，高いリソース効率を実現可能な技術

として，Network Function Virtualization (NFV) が注目されている．NFV 環境を構築する際には，ハードウェアアプライアンスの代替として Linux 上の仮想マシンなどが利用されるため，従来よりも設備投資などに要するコストを削減できる．しかしながら，汎用性の高い Linux は高度に複雑なネットワークスタックを実装しており，多様な機能の積み重ねがネットワークの性能を劣化させてしまっている．ゆえに現状では，Linux が提供する通常のネットワークスタックを用いて，NFV が要求する水準の性能を達成することは難しい．

こういった性能面での問題に対応するために，ユーザ空間で高速なネットワーク処理を実現する手法として，DPDK [1] といった，カーネルをバイパス技術が提案されている．しかしながら，Linux カーネルをバイパスする場合，既存のネットワークスタックが備える豊富で成熟した機能群を活用することはできず，専用のネットワーク機能は，各々の API に合わせてフルスクラッチで実装しなければならない．さらに DPDK は， CPU を占有することで高い性能を実現しており，占有した CPU はアイドル状態に遷移しないためリソース効率が悪く，電力の消費を増加させてしまう[*1]．またユーザ空間でバイパスする代わりに，独自のカーネルモジュールを作成することで，既存のネットワークスタックをバイパスする方法も考えられるが，システム全体のクラッシュや異常動作を引き起こす可能性があるため，商用サービスでの利用は敬遠される．

以上のような問題を受けて，Linux ネットワークにおける拡張可能な高性能データパスとして，eXpress Data Path (XDP) という機能が実装された [2]．XDP は，Linux が提供する既存のネットワークスタック機能と連携した動作が可能で，アイドル状態への遷移もできることから，リソース効率や消費電力の観点で優れている．加えて，内部で安全性を検証するため，カーネルモジュールと比較して，クラッシュや異常動作を引き起こす可能性は低い．XDP では，extended Berkley Packet Filter (eBPF) の仕組みを応用し，Linux に同梱されているネットワークデバイスドライバに対して，ユーザが実装したネットワーク機能を，eBPF プログラムとして挿入する．これによって，デバイスドライバでパケットを受信した直後に，ユーザが実装した eBPF プログラムによる処理を行えるため，高速で柔軟なネットワーク機能を実現できる．

eBPF は 2013 年に Linux に取り込まれた機能で，カーネルの内部にある定められた箇所に対して，ユーザが記述した任意の処理を挿入することで，カーネルを柔軟に拡張できる．eBPF の前身となる BPF [3] は，プログラムの記述をアセンブラで行うため，拡張性は限定されていたが，eBPF では C 言語で実装できるよう改良されており，従来よりも大幅に拡張性が向上している．しかしながら現状では，eBPF によるプログラミングは一般的な C 言語に比べ非常に制約が厳しく，また，デバッガが利用できないなど，一般的な C 言語とは異なるノウハウが必要とされるため，実装の難易度は高い．XDP を利用したネットワーク機能の実装も同様で，ネットワーク機能の開発者や運用者が利用するには敷居が高く，未だに Linux カーネルの開発者を含む，ごく一部でしか利用されていない．

1.2 目的

本研究では，XDP によるネットワーク機能作成において，eBPF プログラム作成の困難さを回避あるいは解消し，eBPF 利用時の障壁を低減することを目的とする．

XDP はリソース効率に優れるため，応用例の NFV で仮想スイッチなどに適用できれば，NFV のメリットであるリソース効率の向上などにつながる．本研究の目的が達成された際には，ネットワーク機能の開発者やネットワーク運用者は，容易にネットワーク機能を拡張できるため，NFV のメリットである柔軟なネットワーク構成を享受できる．

1.3 構成

本稿ではまず，研究の背景や，その目的について説明した．この後の 2 章では，本研究の中核を担う eBPF について詳細に説明し，その応用事例や本研究で取り組む課題について述べる．さらに，2 章で述べた課題を解決するために，本稿では 2 つのアプローチを順次検討したので，それぞれのアプローチの詳細とその検討結果を，3 章および，4 章で示す．次いで 5 章では，アプローチの 1 つであるネットワーク機能の隠蔽に関連した技術を紹介し，提案手法との差分を明らかにする．最後に 6 章で本研究についてまとめ，今後の展望について述べる．

2. 問題提起

本章では，本研究で取り組む課題について，eBPF の技術的観点を踏まえた上で問題を提起する．そのためにも，まずは eBPF プログラムが実行されるまでの過程とその構造，さらに応用事例など，前提となる事柄について説明する．

2.1 extended Berkeley Packet Filter

図 1 に eBPF プログラムの挿入について示す．eBPF によってカーネルの機能を拡張する場合，ユーザはカーネルに挿入する eBPF プログラムと，eBPF プログラムを制御するためのユーザ空間アプリケーションを，それぞれ別々に作成する必要がある．

eBPF プログラムは一般的に C 言語で作成され，Clang/LLVM コンパイラを通して eBPF バイトコードへと変換される．変換された eBPF バイトコードをカーネ

[*1] CPU をアイドル状態に遷移させる設定も可能だが，性能は低下する．

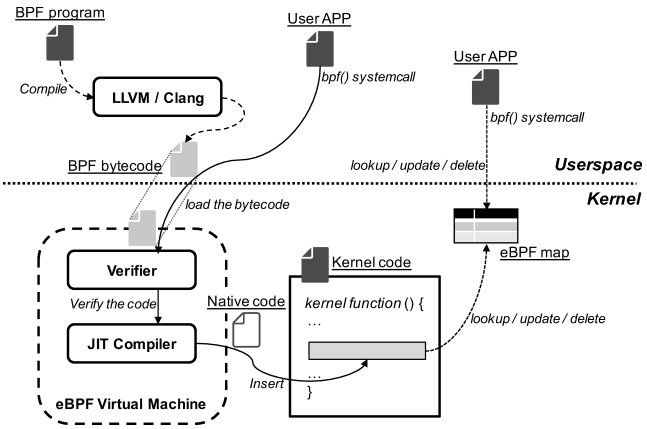

図 1　eBPF プログラムをカーネルに挿入するフロー [4]
Fig. 1　Insertion of an eBPF program into kernel [4].

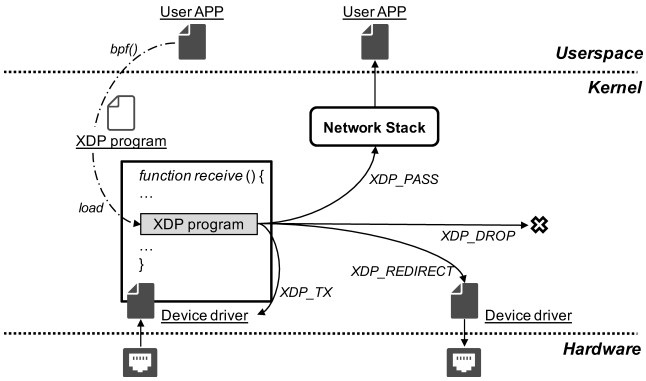

図 2　XDP によるパケット処理のフロー [7]
Fig. 2　Packet processing with XDP [7].

ルに挿入する際には，bpf システムコールが利用され，まずはカーネル内部にある Verifier によって静的に安全性がチェックされる．その後，カーネル内部で JIT コンパイラが有効な場合はネイティブコードに変換され，そうでない場合はカーネル内のインタープリタにより実行されることで，戻り値に基づいた処理を実行することが可能となる．カーネルに挿入される際に Verifier によるチェックが行われるため，eBPF による機能拡張はカーネルをクラッシュさせる可能性が低く，安全性が高いといえる．

ユーザ空間アプリケーションにおいて，ユーザ空間とカーネル間でデータをやりとりする際には，図 1 中に示されるような eBPF map をはじめとするデータ構造が使用されている．eBPF map は任意の型の Key-Value 連想配列であり，ユーザ空間のアプリケーションから bpf システムコールを呼び出すことによって作成され，eBPF プログラムは作成された eBPF map にアクセスをすることができる．

eBPF はカーネル内の様々な箇所に柔軟に機能を挿入できることから，クラウド基盤など複雑な構成における障害解析や，仮想ネットワークの高速化など，幅広く応用事例が提案されている．例えば eBPF を適切に利用すると，kprobe [5] で設定したブレークポイントに対して任意の処理を実行することができるため，カーネル内でのデータの蓄積や，返り値に応じた処理結果を返せるようになる．また仮想ネットワークの分野では，1.1 章で説明した高速なネットワーク技術として，XDP の活躍が期待されている．XDP では**図 2** に示すように，カーネル内でパケットを NIC から取り出した直後に，ユーザがあらかじめ実装し，カーネルへと挿入したプログラムによる処理を実行できる．これにより，ハードウェアに最も近い部分で，パケットの書き換えや，破棄，転送，受信などのアクションを決定できるため，仮想スイッチや，高速な DDoS 緩和技術などへの応用が期待されている [6]．

2.2　問題の分析

本節では 2.1 章で述べた eBPF の特徴を踏まえ，実装に関連した eBPF の問題について詳しく述べる．

2.2.1　実装における制約

eBPF は，ユーザが作成したプログラムを挿入することによって，柔軟にカーネルの機能をカスタマイズすることを可能にしている一方で，Verifier によってカーネルをクラッシュさせないよう工夫されている．しかし，この Verifier による静的な安全性の確認は，実装面における厳しい制約の上に成立している．例えば，ループ処理やポインタの値参照は禁止されており，eBPF バイトコードにおける命令数は最大でも 4096 に制限されている．また安全性を確認する際に，コンパイラによる最適化の影響で，正しい判定ができないケースもある．安全でないプログラムが安全と判断されることはないものの，本来ならば安全とされるプログラムですら，安全はでないと判断され，プログラムの挿入に失敗することはあり得る．こういった制約は，eBPF プログラム作成時のユーザビリティを低下させている．

2.2.2　学習コスト

eBPF バイトコードは C 言語からコンパイルされるが，eBPF プログラムの実装時に利用される C 言語は，通常の C プログラムとは部分的に記法が異なっている．例えば eBPF プログラムでは，カーネルポインタを読むことができないため，現状では，代わりにポインタをデリファレンスするためのヘルパー関数を利用することによって，同等の機能を実現するしかない．また前述の通り，eBPF プログラムには命令数の上限が定められており，大規模な機能を実装することができないため，Tail Calls という仕組みを応用して，複数の eBPF プログラムを連結する．このような特殊な作法は学習コストを高めるため，eBPF や XDP を導入する際の障壁となってしまう．

3.　実装の隠蔽

本研究ではまず，難易度の高い eBPF によるネットワー

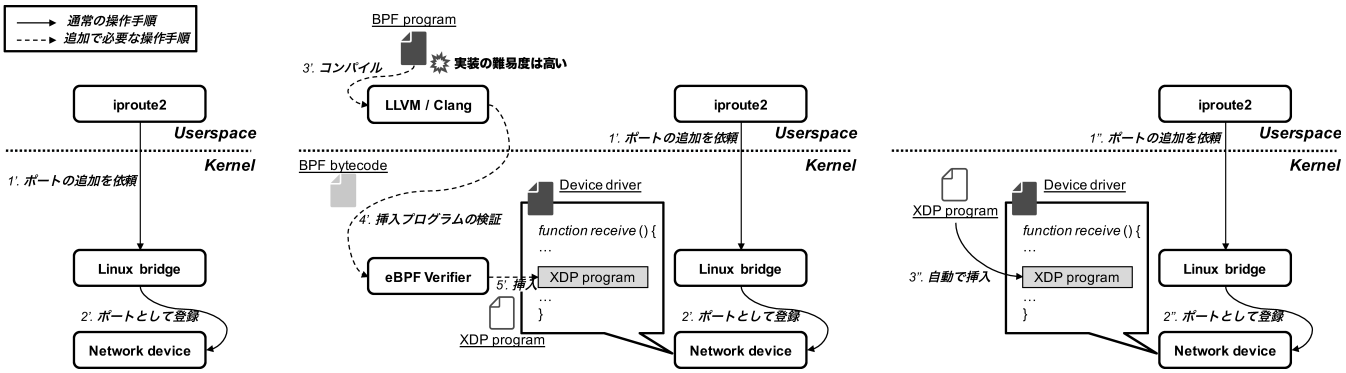

図 3 作成手法の比較．(a) 通常の Linux bridge，(b) 通常の XDP スイッチ，(c) 提案手法
Fig. 3 The flow of adding a new port in (a) the normal Linux bridge, (b) the normal XDP switch, and (c) the proposed method.

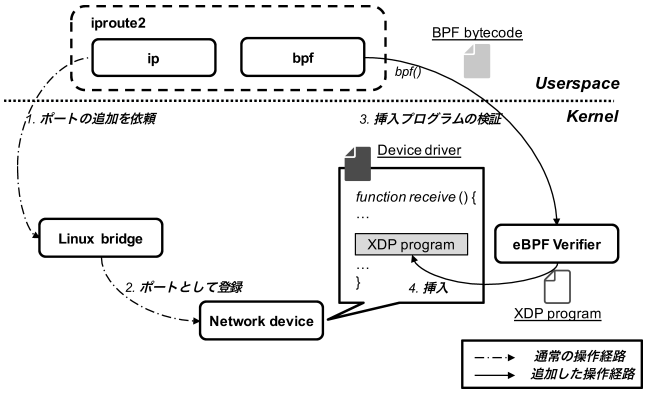

図 4 ユーザ空間における隠蔽
Fig. 4 Concealment in userspace.

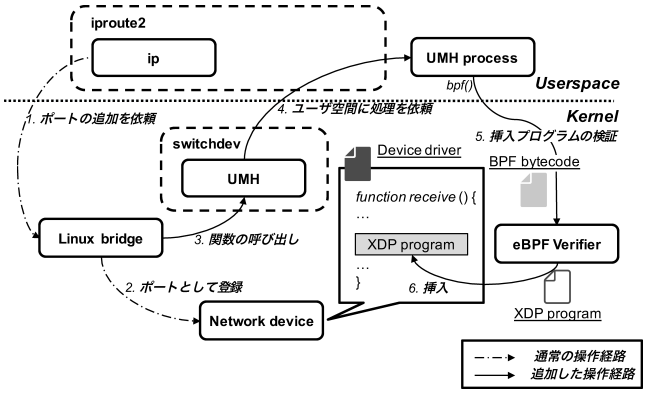

図 5 カーネルにおける隠蔽
Fig. 5 Concealment in kernel.

ク機能の作成を回避するために，2 章で述べた課題を解決するアプローチの 1 つである，実装を隠蔽する方式について検討する．本稿における隠蔽とは，Linux bridge [8] など，既存のカーネルに備わるネットワーク機能の操作で利用される API を通じ，eBPF による機能の挿入を透過的にライブラリもしくはカーネルで行うことを指す．これにより，ユーザは難易度の高い eBPF プログラムを実装することなく，XDP を利用できるようになる．

しかしながら，すでに述べたように，隠蔽はカーネルに現存する機能を対象とするため，新しい機能をユーザが実装するような状況においては有用でない．したがって，こういった状況に対応するためにも，eBPF に不足する機能などを考察し，eBPF そのものにおけるユーザビリティを改善する必要がある．検討した内容については，4 章で詳細に説明する．

3.1 機能要件および設計

本稿で提案するネットワーク機能の隠蔽手法で必要とされる要件は，次の通りである．

1. カーネルを操作する既存の API との互換性
2. カーネルが提供するネットワーク機能を超える高速なパケット処理性能

以上を踏まえ，本稿ではネットワーク機能の一例として L2 スイッチ機能を対象とし，Linux bridge と同じユーザインタフェースを利用して，高速化された機能を既存のインタフェースで隠蔽する．まずは L2 スイッチ機能で方式のフィージビリティを確認し，汎用的なものへと用途を拡張していく．

一例として，**図 3** の左図において，通常の Linux bridge に新しくポートを追加する場合の流れを示す．これに対して，XDP によって高速化された Linux bridge を実現する場合，図 3 の中央図に示すように，通常の工程に加えて，自身で eBPF プログラムを実装し，カーネルへとプログラムを挿入しなければならない．eBPF によるネットワーク機能の実装は容易ではないので，ユーザが eBPF プログラムをすることなく XDP を利用できるように，図 3 の右図のような手法を検討し，既存の手法と同様の操作によって，事前に用意されたプログラムを自動的にカーネルに挿入することで改善を図る．

3.2 実装

Linux bridge のインタフェースである netlink [9] は，iproute2[*2] [10] が提供する ip コマンドおよび libnetlink ラ

[*2] ユーザ空間でネットワークを管理するためのパッケージ．

イブラリによって操作されるため，iproute2 による Linux bridge への操作と同様の操作で，XDP によって高速化された bridge を操作できるように，iproute2 またはカーネルに機能を追加する．通常の Linux bridge では，iproute2 によってコマンドが実行されると，netlink インタフェースを経由して，カーネルに操作を指示する．この通常の操作を踏まえ，ユーザ空間もしくはカーネルの内部で eBPF によるプログラムの挿入を隠蔽し，3.3 章でそれぞれの方式を比較する．

3.2.1 ユーザ空間における隠蔽化

ユーザ空間における隠蔽方式では，iproute2 の内部で隠蔽を実現し，従来の通り netlink 経由で Linux bridge に操作を指示したあと，これに加えて XDP により実装された L2 スイッチへの操作を行う．例えば，高速化された Linux bridge にポートを追加する場合，**図 4** の実線で示されるように，カーネルによるポートの追加処理が終わると，iproute2 は続けて bpf システムコールを呼び出し，実装された L2 スイッチ機能を eBPF プログラムとしてカーネルへと挿入する．この際，挿入される eBPF プログラムは，iproute2 内のソースコードの一部として管理されており，iproute2 をビルドするのと同時にコンパイルされ，運用システムに配置されるものとする．システムコールの呼び出しに伴って，eBPF バイトコードは Verifier で検証され，XDP プログラムとしてデバイスドライバへと挿入される．

3.2.2 カーネルにおける隠蔽化

次いで，**図 5** を用いてカーネルの内部で機能の挿入を隠蔽する方式について示す．実現にあたって様々な手法が考えられるが，例えば，Linux bridge に直接 eBPF バイトコードを埋め込むことによって一連の処理を隠蔽する場合，Verifier による安全性の確認が行われずに，カーネルを意図せずクラッシュさせてしまうおそれがある．こういった危険性を排除するためにも，ユーザ空間で隠蔽する方式と同様に，bpf システムコールによってネットワーク機能を挿入するアプローチを検討した．Linux では，ユーザ空間の機能をカーネルが利用できるようにユーザ・モード・ヘルパー（UMH）を提供しており，提案する手法では，この UMH を利用することでユーザ空間のプロセスに処理を依頼する．カーネルからの通知に伴って，ユーザ空間のプロセスは bpf システムコールを呼び出し，あらかじめコンパイルされた eBPF バイトコードを安全に挿入する．ここで挿入される eBPF プログラムは，Linux カーネルのソースコードの一部として管理されており，Linux カーネルをビルドするのと同時にコンパイルされ，運用システムに配置されるものとする．

しかしながら，上述した内容を Linux bridge に直接的に実装する場合，既存のソースコードを大幅に変更する必要があるため，本来の機能に悪影響を与えてしまう可能性

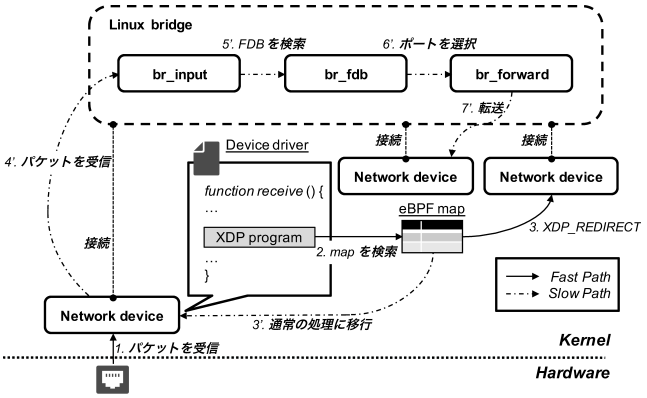

図 6 提案手法におけるパケット処理のフロー
Fig. 6 Packet processing in the proposed XDP bridge.

がある．そこで本稿では，switchdev [11] を応用した関数のオフロード機能によって，Linux bridge のソースコードに大きな変更を加えないように配慮する．switchdev は，ハードウェア毎に専用のドライバを作成することによって，カーネル内のネットワーク機能をハードウェアにオフロードするフレームワークで，現在では，Rocker [12] をはじめとする様々なネットワークスイッチ用に switchdev に対応したドライバが実装されている．既存の Linux bridge においても，カーネルからハードウェアに処理をオフロードする際にはこの switchdev が利用されており，UMH によるシステムコールの呼び出しにこれを応用する．つまり，ハードウェアスイッチに Linux bridge の処理をオフロードする代わりに，UMH でユーザ空間に対して，XDP 関連の処理を委譲する機能が実装された switchdev ドライバを作成する．既存の機能を有効に活用することによって，Linux bridge におけるソースコードの書き換えを最小限にとどめつつ，実装の隠蔽を実現する．

3.3 方式の比較による考察

3.2 章で提案した隠蔽方式それぞれを実装した結果，高速な転送機能を備えた L2 スイッチのフィージビリティを確認できたため，様々な観点から各方式について比較を行い，検討する．本節では，それぞれの項目について詳細に解説し，最終的にどちらの方式を採用すべきか検討する．

3.3.1 転送性能

いずれの実装方式においても，Linux bridge のポートとして登録されたインタフェースのデバイスドライバには，同じ XDP プログラムが自動で挿入されることになる．したがって，性能面における両方式の差異はほとんどないと考えられる．

それぞれの方式では，高速な L2 スイッチの上位レイヤとして通常の Linux bridge を同時に作成しており，高速な転送処理だけでなく，Linux bridge が提供する豊富な機能を最小限の実装で実現している．つまり，フラッディングやラーニングといった，高速化が必ずしも重要でないパ

表 1　実装方式の比較
Table 1　Comparison of proposed methods.

	フィージビリティ	転送性能	実装の容易性	既存アプリケーションとの親和性	機能の拡張性
ユーザ空間における隠蔽方式	✓	✓	✓		
カーネルにおける隠蔽方式	✓	✓		✓	✓

ケット処理は，上位に位置する通常の Linux bridge に委譲することが可能なため，高性能が必要とされるパケット処理時のオーバヘッド削減に貢献できる．**図 6** に示すように，XDP の L2 スイッチは内部に eBPF map で作成された FDB のキャッシュを保持する．この実装では，特に宛先 MAC アドレスが限定されている場合にはラーニングを無効化し，静的な FDB エントリのみを扱うようにすることで eBPF map の更新が不要になるため，非常に高速に動作することが期待できる．

3.3.2　実装の容易性

3.2 章で説明したように，提案した方式は bpf システムコールを利用して安全性を向上させるという性質上，ユーザ空間で隠蔽する方式は比較的に実装がしやすい．これに対して，カーネルで隠蔽する方式では，UMH を経由してユーザ空間のプロセスとやりとりする際の，排他制御およびトランザクション管理について検討しなければならない．また，ユーザ空間のプロセスが予期せず停止あるいは終了してしまった場合も考慮する必要もあるため，カーネルで隠蔽する方式は実装の難易度が高い．

3.3.3　既存アプリケーションとの親和性

既存の API を通じて機能を隠蔽する際には，方式に関わらず，何らかの方法によって，XDP による高速化を行うか否かについて，ユーザが指示をする必要がある．

本稿で提案したユーザ空間における隠蔽手法は，XDP の高速化を指示するインタフェースとして，iproute2 にコマンドオプションを追加している．すなわち，既存の iproute2 を利用するプログラムで XDP による高速なパケット処理機能を利用するためには，新たに iproute2 のオプションを追加する変更が必要となる．ゆえに，例えば OpenStack[*3] [13] において，提案手法に対応する場合は，OpenStack のソースコードを修正し，この方式に対応したバージョンをシステムに配備する必要がある．ユーザ空間における隠蔽方式は影響波及範囲が大きいため，既存アプリケーションとの親和性は低い．

一方で，カーネルレイヤで機能を隠蔽する際には，Linux bridge で利用される netlink インタフェースではなく，Linux bridge に追加するポートへの設定によって実現が可能である．具体的には，ethtool[*4] の -K オプションを利用し，ポートに接続されたデバイスの feature 設定に対して，XDP スイッチの高速化に関わる項目を追加する．この方式では，システム起動時に実行される rc.local などに対して，ethtool の実行を追記するだけで高速化を指示できるため，OpenStack など，既存のプログラムに影響が及ぶことはない．したがって，カーネルレイヤで隠蔽する方式は柔軟性が高く，それに応じて，既存アプリケーションとの親和性も高い方式であるといえる．

3.3.4　機能の拡張性

ユーザ空間で実装の隠蔽を行う場合，今回のように利用するライブラリごとに実装を施す必要がある．本稿で利用した iproute2 の他にも，pyroute2[*5] など，Linux bridge を操作するツールは多様化しており，これらすべてに同様の実装をしていくのは効率的ではないと考える．それに対して，カーネルレイヤで隠蔽する際には，ユーザ空間で使用されるインタフェースの影響を受けないため，最小限の実装で目的とする機能を実現できる．

3.3.5　検討結果

以上の内容についてまとめた結果を，**表 1** で示す．この表から，カーネルで実装を隠蔽する方式は実装に難があるものの，ユーザ空間における隠蔽方式と比較して，高い親和性と拡張性を提供できることがわかる．本稿では，カーネルで実装を隠蔽する方式における，排他制御やトランザクション管理を除いた実装のフィージビリティを確認したので，引き続き，こういった実装が困難な処理の実現が可能か検討を行う．それが完了し次第，カーネルで実装を隠蔽する方式を Linux カーネルコミュニティに提案し，細かい改善点などについて議論していく．

4.　ユーザビリティの改善

3 章で説明した上位レイヤにおける実装の隠蔽は，ユーザが eBPF プログラムを実装することなく，高速なネットワーク機能の提供を実現している．しかしその一方で，カーネルや周辺ライブラリの API を再利用するという性質上，カーネルに具備されたネットワーク機能以外には対応することができない．

上記の理由から，より広範に渡って eBPF を活用するには，eBPF プログラムを利用者自身が開発しなければならないため，eBPF 自体のユーザビリティをも改善する必要がある．そのためにもまずは，隠蔽する機能の実装を通して eBPF のユーザビリティにおける課題を抽出し，それらを整理する．

*3　クラウド管理ソフトウェア．
*4　イーサネットデバイスの各種設定を行うツール．
*5　Python の netlink ライブラリ．

表 2　ヒューリスティック評価を活用した課題の抽出
Table 2　Extraction results of issues by applying heuristic evaluation.

評価項目	代表的な課題	新規 / 総数
Visibility of system status	eBPF プログラムのデバッグ機能が不足している	1 件 / 1 件
Match between system and the real world	Verifier のエラー出力が理解しにくい	1 件 / 1 件
User control and freedom	–	–
Consistency and standards	通常の C 言語の機能が利用できない	0 件 / 4 件
Error prevention	eBPF map に対してユーザ空間から排他制御を行えない	1 件 / 2 件
Recognition rather than recall	挿入した eBPF プログラムのソースコードを表示できない	0 件 / 1 件
Flexibility and efficiency of use	カーネルツリー外では eBPF プログラムのビルドが難しい	2 件 / 2 件
Aesthetic and minimalist design	繰り返し処理ができない	0 件 / 1 件
Help users recognize, diagnose, and recover from errors	Verifier ではエラーの対処方法が示されない	1 件 / 1 件
Help and documentation	カーネルドキュメント [14] の内容は充実していない	1 件 / 1 件

表 3　抽出した代表的な課題の優先度
Table 3　The priority levels of some extracted issues.

代表的な課題	優先度
eBPF プログラムに関わるデバッグ	高
eBPF map に対するユーザ空間での排他制御	高
C 言語による実装の制約	中〜高
Verifier による誤判定	中〜高
カーネルツリー外での eBPF プログラムのビルド	中
ドキュメンテーションの充実	中
ヘルパー関数の機能不足	低〜中
各ドライバにおける XDP のサポート	低

4.1　課題の分析

本節ではユーザビリティの改善に向けて，まずは，隠蔽化の実装を通じて課題を発見する際に利用した，課題の抽出手法および，その抽出結果について説明する．その後，関連する論文 [15] や OSS のドキュメント [16] ですでに報告されている課題を含めて整理した結果を踏まえ，優先して改善すべき課題について解説する．

隠蔽化の実装を通して抽出した課題の網羅性を向上させるために，本研究では，Nielsen らによって提案されているヒューリスティック評価 [17] を活用した．**表 2** はヒューリスティック評価の項目ごとに，発見した代表的な課題の内容と，抽出した課題の件数を示している．件数のうち，隠蔽機能の実装を通して抽出された課題は，新規に抽出された課題として総数とは別に記載している．なお，User control and freedom[*6] の項目には課題が見つからなかったが，eBPF は Verifier により無限ループなどの望ましくない状態の発生を防止しているため，該当する課題は存在しないと考える．結果として，User control and freedom を除いた，全ての項目において課題を抽出した．

4.2　課題の整理

続いて，抽出した課題の優先度について検討する．本稿では，次に示すルールに基づいて優先度を付与する．

i. 他の課題を解決することで課題自体がなくなるものや，利用頻度を鑑みて実質的に不要と考えられるものは優先度を低とする．
ii. すでに他の誰かが取り組んでいるものに関しては，積極的に取り組む必要がないため，優先度は中とする．
iii. 一定の複雑さや学習コストを犠牲に制限などを回避する手法があるものは，中〜高を設定する．
iv. i 〜 iii に当てはまらないの課題は，すべてクリティカルな問題であると捉え，優先度は高と定めている．

課題解決の優先度をまとめた結果について，**表 3** を用いて説明する．本研究では，優先度を低としたものは対象外とし，中より優先度の高い課題を改善していくものとする．優先度が高とされている課題には，今後も継続して重点的に取り組み，中〜高の課題に関しては，コミュニティに内容を報告した上で，連携して改善を図る．一方で，優先度が中に設定されている課題は，自身では取り組まずに，コミュニティと連携することによって解消を目指す．

上述した優先度を考慮すると，eBPF プログラムにおけるデバッグや，eBPF map における排他制御といった課題は，他の課題よりも優先的に解決されなければならないことがわかる．本節ではこれらの優先度の高い課題について，詳細を述べる．

4.2.1　eBPF プログラムにおけるデバッグ

システムや作成したプログラムの状態を正確に把握するには，適切なフィードバックを通知するようなデバッガが必要とされるが，eBPF プログラムに対応した gdb のようなデバッガは存在していない．eBPF プログラムはカーネルに組み込まれて動作するため，カーネルで提供される kprobe や perf といった，デバッグ機能の対応が不可欠である．kprobe はカーネルの内部にブレークポイントを設定する機能だが，現状では，eBPF プログラムの内部に設定することはできない．一方で，カーネル付属のプロファイラとして利用される perf は，関数単位で CPU の使用率を算出できるが，ユーザが挿入した eBPF プログラム

[*6] ユーザが望まない状態に陥った際に，その状態から抜けられる仕組みがあるか．

に関しては関数名が出力されず，ソースコードとの対応関係がわかりづらい．現在では，eBPF のエントリポイントとなる関数名を表示する機能などが実装されているが [18] [19]，その関数内から呼ばれる関数名は表示されないなど，機能としては不十分である．

現在の Verifier には，eBPF プログラムの挿入が失敗した際に，エラーが発生した箇所に該当するソースコードを表示する機能 [20] など，出力されるエラーをわかりやすく改善する機能が実装されている．しかしながら，エラーの内容を理解するには，eBPF で利用される仮想マシンのレジスタがソースコード内のどの変数に相当するか把握するなど，eBPF とコンパイラの内部実装に関して高度な知識が必要とされる．未だに変数とレジスタの対応関係は出力されないため，Verifier から出力されるエラーの内容を推定することは，依然として困難である．

通常の C プログラムであれば，デバッグ機能を充実させるためには，コンパイラが出力する DWARF などのデバッグ情報を活用するのが一般的だが，eBPF においては，デバッグ情報として BPF Type Format (BTF) という形式が利用されており [21]，上述したデバッグ機能の改善にも応用されている．しかしながら，LLVM における BTF の出力機能も，それを活用するカーネル側の実装も現状ではまだ部分的である．

上記に示すとおり，eBPF プログラムにおけるデバッグ機能には未だ改善されていない課題も多く，こういった課題は XDP の実用化を妨げてしまう．不足しているデバッグ機能が充実することによって，Verifier の仕様など，余剰領域への学習を削減できるため，まずは，デバッグ機能における実用性の向上を図る．

4.2.2 eBPF map における排他制御

Software Defined Networking (SDN) のような環境では競合状態が起こり得るため，2.1 章で説明した eBPF map を応用する際には，排他性を保証するような機構が必要とされる [15]．例えば L2 スイッチでは，ユーザ空間のコントローラが MAC アドレステーブルのエントリを参照し，古いエントリを削除するが，排他制御が正しく行われない場合，直前にデータプレーンがそのエントリを利用してタイムスタンプを更新したとしても，コントローラがそれに気付けずに誤ってエントリを削除してしまう可能性がある．

こういった問題を受けて，eBPF map の原子性を向上させる機能として，現在ではスピンロックによる制御機構が実装されている [22]．これにより，未だ機能面での制約はあるものの，eBPF プログラム間においては，格納されているデータの整合性が担保されるようになった．しかしながら現状では，ユーザ空間のプロセスが，上述したような read-modify-write といった一連の処理を不可分に実行するための機能は実装されていない．したがって，現在の実装は排他性を完全に保証するものではないため，ユーザ空間のプロセスと eBPF プログラム間における排他制御に関しては，引き続き方式について検討する必要がある．

4.3 課題抽出のまとめ

本章で抽出した課題を解決することによって，ネットワーク分野における eBPF の利用に関しては，ユーザビリティの問題が改善される．その結果 1.2 章で述べたように，ネットワーク機能の開発者やネットワーク運用者は，容易にネットワーク機能を拡張できるようになることが期待される．

5. 関連技術

本章では，既存のインタフェースを利用して機能を隠蔽する技術として，BPFilter [23] を紹介し，本研究との差分について示す．

BPFilter は，iptables[*7] のインタフェースを利用して eBPF バイトコードを挿入することで，フィルタリング機能における性能の向上を目的としている．具体的には，カーネルモジュールから分離したユーザ空間のプロセスによって，構文解析された iptables の命令が eBPF バイトコードに変換され，任意のポイントでフィルタリングルールとして挿入されるような手法が提案されている．

しかしながら BPFilter では，UMH を利用してユーザ空間のプロセスに処理を委譲することはできるものの，XDP でパケット処理の高速化を実現する機能は採用されていないため，現状では正常に動作しない．以前には，XDP を利用して逐次的にルールを走査するような実装が RFC として提案されたが，大量のルールを挿入した際には，性能が大きく劣化してしまうことが予測される．

また，現在の BPFilter は iptables の内部でフックし，そこから UMH で直接的にユーザ空間にアクセスするような実装がされており，他のネットワーク機能との互換性は低い．それに対して，本研究で提案した方式では 3.2.2 章で述べたように，ネットワーク機能をオフロードするインタフェースとして，switchdev を応用することによって隠蔽機能を実現している．汎用の switchdev インタフェースは拡張性が高く，L2 スイッチ以外のネットワーク機能に対しても適用が可能である．

6. まとめ

6.1 本稿のまとめ

本稿では eBPF における応用事例の中でも，特にネットワーク分野を主な対象とし，難易度の高い実装を隠蔽する方式の提案および比較検討を行った結果，実装の隠蔽はカーネルで行うのが望ましいと判断した．また，隠蔽化の実装を検討する中で，eBPF におけるユーザビリティの課

[*7] Linux に実装されたパケットフィルタリング機能．

題を抽出し，整理した．これらを改善していくことによって，ネットワーク分野におけるユーザビリティの課題は解消が期待される．

6.2 今後の展望

3.3 章で説明したように，今後はカーネルで隠蔽する方式を Linux カーネルコミュニティに提案し，継続的に議論していく．またそれと並行して，4 章で整理した課題の改善に取り組むことにより，NFV 環境におけるネットワーク機能の開発などが容易になっていくと考える．

また 2.1 章でも示したが，eBPF は高速なネットワークだけでなく，障害解析などの分野に対しても応用が期待されている．本稿では，ネットワーク分野における課題のみを整理すべき対象としたため，障害解析の分野に関しては課題をすべて抽出できたとは言い難い．ゆえに，今後は障害解析においても同様にして課題の抽出を行い，それらの改善にも取り組んでいく．

参考文献

[1] Intel, DPDK: Data plane development kit (2014).
[2] Høiland-Jørgensen, T., Brouer, J. D. et al.: The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel, *CoNEXT'18: International Conference on emerging Networking EXperiments and Technologies*, ACM Digital Library (2018).
[3] McCanne, S. and Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture, *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, Berkeley, CA, USA, USENIX Association, pp. 2–2 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1267303.1267305⟩ (1993).
[4] Sharma, S.: Trace Aggregation and Collection with eBPF, Internal tracing meetings, Ecole Polytechnique of Montreal, Montreal (2017).
[5] Linux documentation authors: Linux Tracing Technologies, The kernel development community (online), available from ⟨https://www.kernel.org/doc/html/latest/trace/index.html⟩ (accessed 2019-04).
[6] Bertin, G.: XDP in practice: integrating XDP into our DDoS mitigation pipeline, *Technical Conference on Linux Networking, Netdev*, Vol. 2 (2017).
[7] Monnet, Q.: Introduction to eBPF (for network packet processing), Meetings at frnog (the french network operators group) 28, 6WIND, Paris (2017).
[8] Varis, N.: Anatomy of a Linux bridge, *Proceedings of Seminar on Network Protocols in Operating Systems*, p. 58 (2012).
[9] Salim, J., Khosravi, H., Kleen, A. and Kuznetsov, A.: Linux netlink as an ip services protocol, Technical report (2003).
[10] Kuznetsov, A.: Iproute2: Linux network management, The Linux Foundation (online), available from ⟨https://wiki.linuxfoundation.org/networking/iproute2⟩ (accessed 2019-04).
[11] Pírko, J.: Hardware switches-the open-source approach, Ottawa, Canada, Technical report, Netdev 0.1, The Technical Conference on Linux Networking (2015).
[12] Feldman, S.: Rocker: switchdev prototyping vehicle, *Proceedings of Netdev 0.1* (2015).
[13] Sefraoui, O., Aissaoui, M. and Eleuldj, M.: OpenStack: toward an open-source solution for cloud computing, *International Journal of Computer Applications*, Vol. 55, No. 3, pp. 38–42 (2012).
[14] The kernel development community: The Linux Kernel's BPF Documentation, The kernel development community (online), available from ⟨https://www.kernel.org/doc/html/latest/bpf/index.html⟩ (accessed 2019-04).
[15] Miano, S. et al.: Creating complex network service with ebpf: Experience and lessons learned, *High Performance Switching and Routing (HPSR). IEEE* (2018).
[16] Cilium Authors: Cilium's BPF and XDP Reference Guide, Cilium Authors (online), available from ⟨https://cilium.readthedocs.io/en/latest/bpf/⟩ (accessed 2019-04).
[17] Nielsen, J.: Usability Engineering, Academic Press, San Diego, pp. 115–148 (1994).
[18] Liu, S.: reveal invisible bpf programs, The kernel development community (online), available from ⟨https://patchwork.ozlabs.org/cover/1026789/⟩ (accessed 2019-04).
[19] Liu, S.: perf annotation of BPF programs, The kernel development community (online), available from ⟨https://lore.kernel.org/patchwork/cover/1049975/⟩ (accessed 2019-04).
[20] Lau, M.: verbose log bpf_line_info in verifier, The kernel development community (online), available from ⟨https://patchwork.ozlabs.org/cover/1013039/⟩ (accessed 2019-04).
[21] Borkmann, D., Song, Y., Nakryiko, A. et al.: BPF Type Format (BTF), The kernel development community (online), available from ⟨https://www.kernel.org/doc/html/latest/bpf/btf.html⟩ (accessed 2019-04).
[22] Starovoitov, A.: introduce bpf_spin_lock, The kernel development community (online), available from ⟨https://lwn.net/Articles/776909/⟩ (accessed 2019-04).
[23] Borkmann, D.: net: add bpfilter, The kernel development community (online), available from ⟨https://lwn.net/Articles/747504/⟩ (accessed 2019-04).