SDN環境におけるスライスの独立性検証の高速化

江幡 正樹[†1]　阿部 洋丈[†1]　下條 真司[†2]　伊達 進[†2]　野崎 一徳[†2]

概要：医療などのデータ基盤には、堅牢なセキュリティが求められる。そのセキュリティの 1 つとして、利用組織別にネットワークのスライスを提供する手法が考えられる。ただし、多くのスライスが作成される環境で、各スライスの独立性を管理し続けることは困難である。そこで、本論文では多量のスライスの独立性を高速に検証を提案する。提案手法では、スライスの増加に耐性のある高速な検証を行うために、トライ木を使用してスライスのヘッダの重複を計算する。評価の結果、本提案手法は先行研究と比べてヘッダの重複を多項式時間から線形時間で計算できるようになった。評価環境では、アドレス範囲が 2^{26} 以上であれば、リアルタイム性が達成できた。

キーワード：Software-defined networking, ネットワーク検証, ネットワークスライシング, スライスアイソレーション

1. 序論

現在、日本は超高齢社会を迎えており、医療機関の負担が非常に高い状態にある。これによって危惧されるのが、がんなどの早期治療が重要な疾患の発見の遅れや、誤診や処置ミスの多発化である。こうした将来の問題に早くから対処することは、国民の平均年齢が高い日本にとって非常に重要である。

この問題に対処する試みとして、S2DH[1] という産学連携プロジェクトがある。S2DH は、機械学習による口腔医療の自動化・支援の実現を目指している。その主な活動内容に、舌がんの自動診察や、矯正治療の要否識別がある。このような自動化によって、外来受診や医療処置の高効率化が期待される。

口腔医療の自動化するための人工知能には、顔写真や口腔画像を利用する。ただし、顔写真はもちろんのこと、口腔画像に含まれる歯並びも身元確認に使用できるプライバシー情報である。そのため、S2DH の実現には、医療データを安全に収集・加工する医療用データ基盤が必要である。

医療用データ基盤には、データ提供者 (以下、提供者) に医療機関、データ利用者 (以下、利用者) に大学・企業・研究所が想定される。このとき、それぞれの接続者に同一のネットワークを提供すると、全ての接続者が広い範囲で到達可能性を持ってしまう。この状況では、接続者に悪意のある人物が居た場合、DDoS 攻撃や情報漏えいの危険性がある。そのため、このデータ基盤のネットワークでは、組織別に異なるスライスが提供されていることが理想的である。

医療などのデータ基盤上のスライスは、それぞれ独立性を持っていなければならない。スライスの独立性とは、スライス間でパケットが混同しないことを意味する。独立性を保つためには、ネットワーク管理者がスライスの設定を重複させないよう管理しなければならない。そのため、医療用データ基盤のように多量のスライスが作成される環境では、全スライスの独立性を保つことは困難である。

スライスの障害を検出するための手段に、SDN 環境の検証システムが考えられる。Header Space Analysis[9](以下、HSA) は、多様化・複雑化したルーティングの検証手法を提案した研究である。この HSA の検証の対象には、スライスの独立性の検証が含まれている。HSA の独立性検証では、スライスに割り当てられたヘッダとポートが、他のスライスと重複しているかどうかを計算する。ただし、この手法はネットワークに対して静的な検証を想定しており、計算量が大きい。そのため、動的に作成されるスライスの障害を未然に防ぐことが出来ない。

そこで、本研究では多量にスライスが作成される環境に対して、スライスの独立性を高速に検証する手法の提案を目的とする。本研究の提案手法では、スライスの増加に耐性のある高速な検証を行うために、トライ木を使用してスライスのヘッダの重複を計算する。評価の結果、本提案手

[†1] 筑波大学
[†2] 大阪大学

ⓒ 2019 Information Processing Society of Japan

法はHSAと比べてヘッダの重複を多項式時間から線形時間で計算できるようになった。ただし、検証時間がヘッダのアドレス範囲に大きく依存することも確認された。評価環境では、アドレス範囲が 2^{26} 以上であれば、新規スライスの98パーセンタイルに1ms以下で検証ができ、リアルタイム性が確認できた。

2. 関連研究

2.1 ネットワークスライシング

ネットワークスライシングを実現する代表的なプロトコルには、VLANとMPLSがある。これらのプロトコルは、パケットに付与された識別子（VLAN ID、Label）を使用してスライス別の制御を行う。各スライスでは、出力ポート、QoS(Quality of Service)、ACL(Access Control List)を設定できる。これによって、スライスごとに異なるサービスとフォワーディングを提供することができる。

FlowVisor[4]は、データプレーンに複数のコントロールプレーンを接続するプロキシシステムである。SDNを実現する代表的なプロトコルOpenFlow[2]では、コントロールプレーンとデータプレーンを1対多で接続する。これに対しFlowVisorは、プロダクション環境とテスト環境を1つの物理ネットワークで実現するために、コントロールプレーンとデータプレーンを多対多で接続するデザインを提案した。コントロールプレーンが制御するデータプレーンはトポロジ、帯域幅、ヘッダなどの条件を設定できる。そのため、FlowVisorはテスト環境の構築だけでなく、コントロールプレーン別にスライスを提供することもできる。

FlowSieve[5]は、通信相手別に柔軟なアクセス制御を提供するメカニズムである。このメカニズムは、アクセス制御の設定にRBAC(Role-based Access Control)を使用する。RBACは、ロール単位でアクセス権限を割り当てるセキュリティポリシーである。このロールは、1つの通信相手に複数割り当てることができる。これによって、複数のスライスと接続するアクセス制御を円滑に提供できる。

2.2 ネットワーク検証

SDN環境では、コントローラのバグによってネットワークに障害が発生する。これを回避するために、コントローラのアプリケーションのプログラムを検証するという手法が考えられる。しかし、アプリケーションはチューリング完全なプログラムで記述されるため、バグの有無は一般的に決定不能である。また、アプリケーションを記述可能な言語はCをはじめ、JavaやRuby、Python[3]など多岐に渡り、それぞれに対応することは困難である。そのため、ネットワーク検証の関連研究[6][7][8][9][10][12]は、コントローラが生成する制御メッセージを元に検証を行っている。これによって、チューリング完全の問題とプログラミング言語の互換性の問題を回避した検証ができる。

ここでは、検証研究の中でもスライスの独立性を検証対象に含むHeader Space Analysis(以下、HSA)[9]の説明をする。Header Space Analysis(HSA)[9]は、多様化・複雑化したルーティングの静的検証を行うフレームワークを提案した研究である。現在のネットワークでは、プライベートIPとグローバルIPを変換することでアドレス数の制限を回避するNATや、柔軟なルーティングを実現するMPLSが利用されている。こうした複雑なヘッダ変換・転送方法があるネットワーク環境では「あるホスト間が正しく通信できるか」や「任意のパケットが正しくルーティングされているか」を正しく管理することは非常に難しい。そこで、HSAは将来のネットワーク管理の複雑化を回避するために、ヘッダフィールドの書き換えや、複数のプロトコルの転送処理に対応している。

HSAの検証は、ネットワークの幾何モデルに基づいて行われる。ここでは、スライスの独立性検証に関連するヘッダ空間 H、ネットワーク空間 N、スライスの幾何モデルの定義を説明する。HSAでは、スイッチの転送処理に使われるビット列のヘッダ空間を、$H = \{0, 1, x\}^L$ と定義する。L はヘッダフィールドのサイズを表す。例として、L はIPv4アドレスなら32、IPv6アドレスなら128となる。ネットワーク空間は、デバイスが持つポートの識別子とヘッダ空間の組み合わせ　$N = \{0, 1\}^L \times \{1, ..., P\}$　と定義する。これによって、ネットワーク内の各リンクを流れるパケットが表現される。スライスは、ネットワークを仮想的に分割したものであるため、ネットワーク空間の部分集合で表される。例として、スライスはヘッダ空間をIPアドレスとした場合 $\{10.0.1.x \times \{1, 2, 3\}\}, \{10.0.x.x \times \{2, 3, 4\}\}$ となる。

HSAのスライスの独立性検証は2種類ある。1つは、新しくスライスが作成された時に、既存のスライスに対して独立かどうかを検証する。もう1つは、デバイスにヘッダの書き換えが設定された時に、他スライスにパケットがリークしないことを検証する。本研究は前者のケースのみ想定する。そのため、ここでは前者の検証の説明をする。前段落で、スライスはネットワーク空間 N の部分集合で表されると説明した。そのため、スライスの重複は、ネットワーク空間の部分集合の重複として計算される。この重複の有無によって、スライスの独立性が検証される。

ネットワーク空間の比較方法を具体的に説明する。2つのスライス a, b があるとき、それぞれのネットワーク空間 $N_a, N_b \in \mathrm{N}$ を次のように表す。

$$N_a = \{(\alpha_i, p_i)]_{p_i \in P}\}, N_b = \{(\beta_i, p_i)]_{p_i \in P}\} \quad (1)$$

α, β は、それぞれ N_a, N_b のヘッダ空間であり、$p_i \in \mathrm{P}$ は各スライスが使用するポートを表す。この時、2つのスライスが重複していなければ、任意のポートでヘッダの重複がないため、次が成り立つ。

$$\bigcup_{p_i \in P} \alpha_i \cap \beta_i = \phi \quad (2)$$

一方で、2 つのスライスが重複していれば、あるポートとそのヘッダ空間が重複しているので、スライスの積は次のように表される。

$$N_a \cap N_b = \{(\alpha_i \cap \beta_i, p_i)]_{p_i \in N_a \& p_i \in N_b}\} \quad (3)$$

HSA の実装 [11] では、ヘッダ空間の重複計算にビット演算 (AND 演算) を使用する。そのために、ヘッダ空間の実装では $\{0, 1, x\}$ を 2 ビットの $\{01, 10, 11\}$ に対応付け、00 を z としている。z はヘッダ空間に存在しない値を表しており、いずれかのビットにこの値が存在していれば、そのヘッダ自体が存在できないものとして扱う。この実装によって、ヘッダ空間における $x \cap 1 = 1$ や $0 \cap 0 = 0$、$0 \cap 1 = z$ をビット演算で容易に計算できる。

ただし、HSA はスライスの独立性の検証で、スライス a, b が持つヘッダ α, β に対して $|\alpha| \times |\beta|$ 回の計算を行う (2 重 for ループを使用)。よって、スライスのポートの重複が多い場合には、ヘッダの重複計算に莫大な計算コストが掛かる。

3. 提案手法

HSA のスライスの独立性検証は、スライス数とそのヘッダ数の増加に応じて検証時間が大きく上昇してしまう問題がある。これは、検証対象のスライスのヘッダと、ポートが重複するスライスの全てのヘッダの重複を計算しているためである。例として、ヘッダ α が 10.0.0.0 ∼ 10.0.255.255、ヘッダ β が 192.168.0.0/16 のアドレスを持つ場合を想定する。この場合に HSA は、ヘッダ α と β の重複を計算するとき、$2^{16} \times 2^{16} = 2^{32}$ 回の計算を行う。しかし、実際にはそれぞれのヘッダは 1 ビット目で異なっている。そのため、この点を考慮できればヘッダの重複計算の殆どを省略することができる。

全く異なるヘッダの比較を省くための手法として、VeriFlow[6] と Delta-net[8] などの木構造によるヘッダの管理が考えられる。VeriFlow のヘッダの検索は、500 万の IP アドレス (v4) のエントリに対して 0.1ms 以下で実行できている。これは、トライ木の検索が長さ L を持つ任意のヘッダに対して、最悪の場合でも L 回で実行できるためである。そのため、トライ木はヘッダ数の増加によって検証時間が増加することが殆ど無い。

そこで、本研究ではスライスのヘッダの重複計算にトライ木を使用した検証手法を提案する。これによって、スライス数・ヘッダ数の増加に影響を受けづらい高速な検証を行う。

ヘッダの重複計算に使用するトライ木の定義を、図 1 に示す。各ノードは VeriFlow と同様 $\{0, 1, *\}$ の分岐を持ち、それぞれヘッダフィールドの値に対応する。ノードの値には、スライスの集合が対応する。スライスの集合には、ノードの *prefix* とヘッダが等しいスライスの識別子のみ挿入する。*prefix* とヘッダが等しいとは、値だけでなく、プレフィックス長も等しいことを意味する。ただし、最後が $*$ の *prefix* に対しては、プレフィックス長が *prefix* 以上に長いヘッダも等しいものに含む。

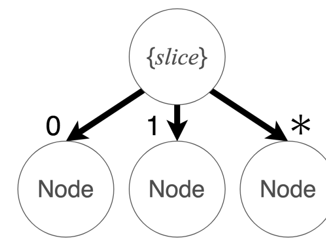
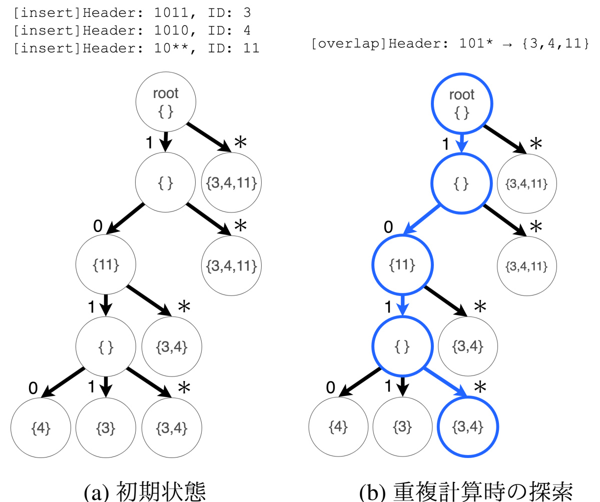


図 1: トライ木の定義



(a) 初期状態　(b) 重複計算時の探索

図 2: トライ木によるヘッダの重複計算の例

このトライ木を用いてヘッダの重複計算を行う方法を、長さ 4 のヘッダを例に説明する。はじめに、トライ木は 3 つのスライス $\{(header : 1011, id : 3), (header : 1010, id : 4), (header : 10{*}{*}, id : 11)\}$ で初期化されているとする (図 2a)。このとき、各スライスの識別子はヘッダに対応するノードだけでなく、そのノード以前にある $*$ のノードにも挿入される。次に、$101*$ のヘッダに重複するスライスを計算する。この計算には、$101*$ のノードの経路上にある値の和集合をとる（図 2b)。これは $1010, 1011$ のような $101*$ で表現されるヘッダだけでなく、$10{*}{*}$ のように検証対象のヘッダよりプレフィックス長が短いものとの重複を考慮するためである。このヘッダの重複計算を行う擬似コードを Algorithm 1 に示す。

本研究のスライスのの独立性検証は、先にヘッダの重複を計算し、後にポートの重複を計算する。HSA の独立性の検証では、先にポートの重複を計算し、そのとき重複したスライスにのみヘッダの重複計算を行っていた。しかし、ポートの重複計算は、全てのスライスに対して計算する必要があり、スライス数に応じて計算量が増加する。一方で、

Algorithm 1 overlap($trie, header$)

```
1: node = trie.head
2: slices = set of a slice
3: h = header.value
4: len = header.length
5: while len > 0 do
6:     slices.insert( node.value )
7:     if h & (1ULL ≪ (header.size-1)) then
8:         node = node.one
9:     else
10:        node = node.zero
11:    end if
12:    len = len - 1
13:    h = h ≪ 1
14: end while
15: slices.insert( node.wild )
16: return slices
```

Algorithm 2 slice_isolation($trie, slice$)

```
1: overlap_slices = set of a slice
2: for header in slice.headers do
3:     h_overlap_slices = overlap(trie, header)
4:     for h_overlap_slice in h_overlap_slices do
5:         for port in slice.ports do
6:             if h_overlap_slice.ports.find( port ) then
7:                 overlap_slices.insert( h_overlap_slice )
8:             end if
9:         end for
10:    end for
11:    trie.insert( header )
12: end for
13: return overlap_slices
```

トライ木を用いたヘッダの重複計算の計算量ならば、スライス数やヘッダ数にあまり影響を受けない。これは、トライ木の探索がヘッダのビット長の回数で収まるからである。そのため、本研究のスライスの独立性検証はヘッダの重複計算を先に行う。これによって、より高速に検証を行うことができると考える。

この計算順序を考慮すると、本検証の計算式は HSA の式 (3) に対して次のように変わる。ここで、p_{a_i}, p_{b_i} は N_a, N_b に属する h_i のポートを表す。

2 つのスライス a, b があるとき、それぞれのネットワーク空間 $N_a, N_b \in N$ を次のように表す。

$$N_a = \{(h_i, p_{a_i})]_{h_i \in H}\}, N_b = \{(h_i, p_{b_i})]_{h_i \in H}\} \tag{4}$$

この時、a, b の重複は次の式によって表される。

$$N_a \cap N_b = \{(h_i, p_{a_i} \cap p_{b_j})]_{h_i \in N_a \cap h_j \in N_b}\} \tag{5}$$

また、スライス a が独立であるとき、次が成り立つ。

$$\bigcup_{N_i \in N} N_a \cap N_i (a \neq i) = \phi \tag{6}$$

この式に沿ってスライスの独立性検証を行う擬似コードを、Algorithm 2 に示す。

4. 評価

本章では、本提案手法のヘッダ数・スライス数別の検証時間 (4.1 節) と、多量のスライスに対する検証時間 (4.2 節)、そしてリアルタイム性 (4.3 節) を評価する。評価用の実験では、HSA と同様の実験を行う。HSA の実験では、FlowVisor[4] によって、スライスごとに送信元/送信先 IP アドレスと使用可能ポートが制限されるネットワークを想定し、新規スライスの独立性を検証する。実験データには、スタンフォード大のトポロジ上でランダムに生成したスライスを使用する。スライスのヘッダに使用するアドレスは指定の範囲でランダムな IP アドレス、ポートは 7 つのポートリストからランダムに選んだものを使用する。実験環境は Ubuntu 18.10、Intel Xeon CPU E5620 @ 2.40GHz、メモリ容量 32GB のコンピュータを使用した。

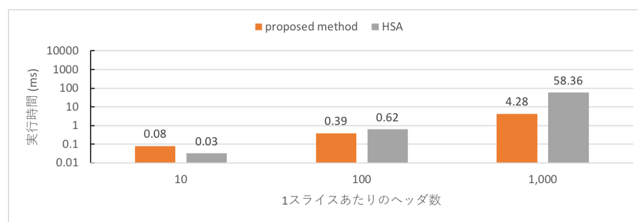
本研究では、関連研究との比較を行うため、HSA の簡易的な再実装を行っている。HSA の再実装は、HSA の公開リポジトリ [11] にあるスライスの独立性を検証するプログラムを参考にした。HSA の検証では、TCP ポートを含んだ検証を行うが、本研究ではこれを省略し、IP アドレスとポートの重複計算のみ実装している。

4.1 ヘッダ数・スライス数別の検証時間

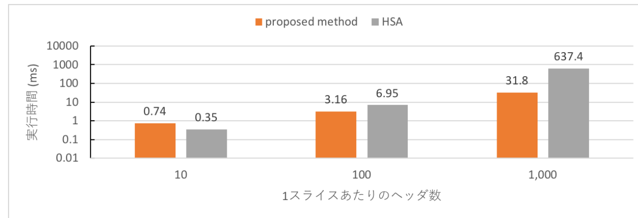
本節では、本提案手法のヘッダ数・スライス数別の検証時間を評価する。この実験は、スライス 1 つあたりに割り当てるのヘッダ数を $10, 100, 1000$ の範囲で変更する。これを、スライス数 $10, 100, 1000$ ごとに切り替えた場合の検証結果を比較する。この実験によって、スライス数・ヘッダ数の変化による検証時間の推移を確認できる。この実験では、再実装した HSA の検証時間も計測した。尚、この実験内容は HSA の評価実験と同様のものである。

図 3 がスライス 1 つあたりの独立性の検証時間をまとめた図である。全体として、各検証手法はスライス数に対して線形時間で検証を終えている。ヘッダ数に関して、HSA は前述通り多項式時間が必要だったが、本提案手法は線形時間で実行できていることがわかる。

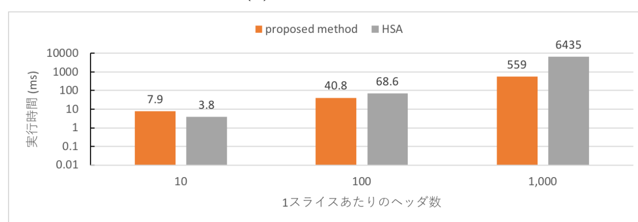
ヘッダ数 10 のケースにおいて、いずれのスライス数でも本提案手法が HSA より低速なのは、ヘッダの重複計算の違いにあると考察する。例として、IP アドレスのヘッダ 171.64.255.255/32 と 171.64.0.0/32 の重複計算を挙げる。本提案手法では、片方のヘッダが既にトライ木に挿入されているとしても、もう 1 つのヘッダを重複計算をするには 32 回のトライ木の探索が必要になる。しかし、HSA では 1 回の AND 演算で計算を終えることができるため、両者には 31 回の計算回数の違いがある。この計算回数の違いが、ヘッダ数の少ないケースで顕著になったと考える。一方で、ヘッダ数が 100 以上のケースでは、計算回数の違い



(a) 10 スライス



(b) 100 スライス



(c) 1000 スライス

図 3: スライス数・ヘッダ数別の検証時間

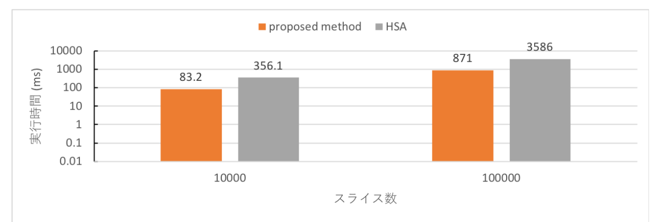
があっても本提案手法の方が高速になる。これは、HSA の全てのヘッダの比較回数が、トライ木の探索回数を上回った結果だと考える。

4.2 多量のスライスに対する検証時間

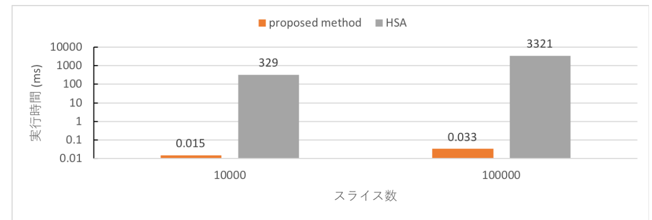
本節では、多量のスライスに対する検証時間を評価するために、前節よりさらに多いスライス (10,000, 100,000) を使用した実験を行ったこの実験では、各スライスに割り当てるヘッダは 1 つに設定した。また、HSA の実験で使用する IP アドレスの範囲は 171.64.0.0/16 と狭く、トライ木のノードの値に偏りが生じる可能性が高い。そこで、この実験では IP アドレスの範囲を 0.0.0.0/0 に変更したケースも検証した。この実験は、前節同様、再実装した HSA の検証時間を計測している。

多量のスライスに対する検証時間の結果を図 4 に示す。それぞれの手法は、前節に引き続き、スライス数に対して線形時間で検証を終えている。HSA は前節で、ヘッダ数が少ない場合に本提案手法より高速に検証を行っていたが、この実験においては HSA より本提案手法のほうが高速だった。また、ヘッダの範囲を広げたケース (図 4b) では、本提案手法は検証を僅か数十 μs で完了できることが確認できた。そのため、本提案手法はスライスのヘッダの範囲に応じて、検証時間が大きく変化する可能性があることがわかる。

図 4a と図 4b の検証時間の違いは、ヘッダが重複するス



(a) IP アドレス 171.64.0.0/16 の場合



(b) IP アドレス 0.0.0.0/0 の場合

図 4: 多量のスライスに対する検証時間

ライス数の違いが原因だと考える。図 4a の実験において、全ての可能な IP アドレスの種類は $2^{17-2} = 131,070$ 個である。そのため、多量にスライスがあるケースでは、ヘッダが重複するスライスの平均的な個数が多いことが考えられる。その場合、ポートの重複計算に多くの計算が必要となるため、全体の検証時間が遅くなる。一方で、図 4b の実験では、IP アドレスの先頭 16bit が $2^{16} = 65,536$ のパターンに分かれる。つまり、図 4 の実験と比べ、ヘッダが重複するスライス数の平均がおよそ $\frac{1}{65536} \approx 10^{-4}$ に少なくなる。これによって、ポートの重複計算の時間が減少し、検証時間に約 10^4 の差が生まれていると考察する。

4.3 リアルタイム性

本節では、本提案手法のリアルタイム性を評価する。ネットワークの検証におけるリアルタイム性とは、コントローラの次の制御メッセージを取得するまでに、それまでの制御の検証が済んでいることを指す。しかし、次の制御メッセージを取得するまでの時間は、環境・コントローラごとに異なるため、明確な基準はない。そのため本研究では、リアルタイム検証システムを提案した VeriFlow[6] の検証性能をリアルタイム性の基準とする。その基準は、ネットワーク制御更新の 98 パーセンタイルに対して 1ms 以下で検証を終えていることである。

本実験では、多量の仮想ネットワークを持つ環境に対してのリアルタイム性を評価するために、100,000 個のスライスを使用した。前節で、本提案手法は IP アドレスの範囲に応じて検証時間が大きく変化することが確認されている。特に、0.0.0.0/0 の範囲でスライスを生成する場合は、数十 μs で検証を完了することができる。一方で、171.64.0.0/16 の範囲の検証は既に 1ms 以上の時間がかかることがわかっている。そこで、本提案手法がリアルタイム性を持つアドレス範囲の限界を探るべく、生成する IP アドレスの範囲

ⓒ 2019 Information Processing Society of Japan

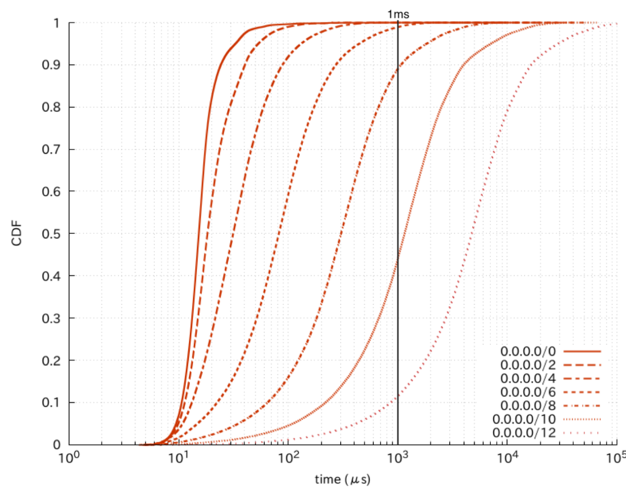


図 5: IP アドレス範囲別の独立性の検証時間

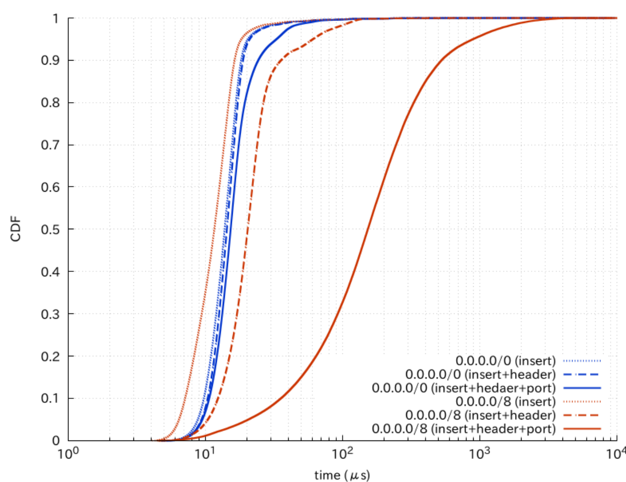


図 6: 独立性検証の各処理時間

には /0 ～ /12 を使用した。

図 5 が、プレフィックス長 /0 ～ /12 の検証時間の CDF グラフである。この結果から、本提案手法はスライスに使用されるアドレスの範囲が広ければ高速に、狭ければ低速になることが確認できる。また、リアルタイム性は生成される IP アドレスの範囲が /6 以上の場合に達成できている。

アドレス範囲が狭まることで検証時間が増加するのは、ヘッダが重複するスライスの平均個数が増加しているからである。提案手法におけるスライスの挿入 (insert)、ヘッダの重複計算 (header)、ポートの重複計算 (port) の時間の CDF を図 6 に示す。このグラフから、insert と header はアドレス範囲に殆ど影響を受けないが、port は大きく影響を受けることがわかる。さらに、ヘッダが重複するスライスの平均個数の推移の結果を表 1 に示す。この表から、平均個数がアドレス範囲に対して指数的に増加していることがわかる。よって、本提案手法はアドレス範囲によって大きな検証時間の違いが生まれる。

表 1: ヘッダが重複するスライス数の平均

/0	/2	/4	/6	/8	/10	/12
0.3	1.3	5.2	21.0	84.8	334.6	1333.6

5. 結論

5.1 まとめ

本研究では、スライスの独立性検証をトライ木を用いて高速化する手法を提案した。評価の結果、本提案手法は HSA と比べてヘッダの重複を線形時間で計算できることがわかった。一方で、検証性能がスライスのアドレスの範囲に大きく依存することも確認された。本提案手法は、アドレスの範囲が広い場合に検証が高速化し、狭い場合に検証が低速化する性質を持つ。リアルタイム性の評価では、100,000 のスライスに対して、アドレスの範囲が 2^{26} 以上 (例えば、0.0.0.0/0 ～ 0.0.0.0/6) であれば、新規スライス更新の 98 パーセンタイルに対してリアルタイム性のある検証を達成できた。そのため本提案手法は、IPv6 でスライスに作成する環境や、プライベートネットワークにおいてリアルタイム性のある独立性検証に期待できる。

5.2 今後の課題

本提案手法は、スライスに使用されるアドレスに偏りがある場合、ポートの重複計算に多くの時間が掛かってしまう。ポートの重複は、集合に対してそのポートの識別値をハッシュで検索する。この計算は、各スライス別に行わなければならないため、ヘッダが重複するスライスが増えるごとに計算時間が増加する。この問題を解決できれば、本提案手法はより幅広い環境に対応できると考える。

本研究の検証では、スライスの ACL(Access Control List) を考慮した検証を行っていない。スタンフォード大学のバックボーンネットワークでは、いくつかの VLAN に対して ACL を設定している。しかし、ACL のプロトコルには IP アドレスだけでなく入出力 TCP ポートなども含まれている。この設定の構文解析と検証の実装が困難だったため、本研究では ACL を考慮した検証を見送った。しかし、実環境で検証を行う際には ACL の対応は必須である。そのとき、全体の検証時間がどれほど増加するのかを確認することは重要だと考える。

謝辞 本研究は JSPS 科研費 17KT0083 の助成を受けたものです。

参考文献

[1] S2DH – 一人ひとりの口（くち）に関する情報をつなぎ新世代の医療・学問分野の創造を目指す, http://s2dh.org/
[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. SIGCOMM Comput.

ⓒ 2019 Information Processing Society of Japan

Commun. Rev., 38(2):69–74, 2008.
[3] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. A Survey of Information-Centric Networking. IEEE Communications Magazine, vol.50, no.7, pp.26–36, 2012.
[4] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In Proceedings 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI'10, 2010.
[5] T. Yamada, K. Takahashi, M. Muraki, S. Date, S. Shimojo. Network Access Control Towards Fully-Controlled Cloud Infrastructure. 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2016.
[6] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in realtime. In Proceedings 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI'13, 2013.
[7] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In Proceedings 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI'14, 2014.
[8] A. Horn, A. Kheradmand, and M. Prasad. Delta-net: Real-time network verification using atoms. In Proceedings 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI'17, 2017.
[9] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In Proceedings 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI'12, 2012.
[10] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In Proceedings 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI'13, 2013.
[11] Hassel-Public, https://bitbucket.org/peymank/hassel-public
[12] H. Yang and S. Lam. Real-time verification of network properties using atomic predicates. In Proceedings 21st IEEE International Conference on Network Protocols, ICNP 2013.

ⓒ 2019 Information Processing Society of Japan