Vol.2020-DPS-182 No.13
Vol.2020-CSEC-88 No.13
2020/3/12

F*言語を用いたネットワークファンクションの設計と実装

細谷 昂平[1,a)] 高野 祐輝[1,b)] 宮地 充子[1,c)]

概要：スマートフォンやパソコンといった多くの機器がインターネットに繋がる社会において，複数の装置同士をつなげるネットワークファンクションはより安全に実装されることが望まれる．実際に，ネットワークスイッチやルーターなどのネットワークファンクションにおいて，モデルの違反によってパケットが欠落するという実行時のバグが発生することで，パケットが目的地に届かないことがある．ネットワークファンクションを検証すると，実際のネットワークのネットワークファンクションが忠実にモデルに従っていることを保証できる．F*言語はプログラム検証を目的とした関数型言語である．本論文では，F*言語を用いて，ネットワークファンクションの一つであるスイッチングハブの MAC アドレスの管理を実現した．この結果フィルタリング機能が検証され，仕様的に誤りのないネットワークファンクションを実現できた．

キーワード：ネットワークファンクション，F*言語，プログラム検証

Design and Implementation of Network Functions Using F*

Kohei Hosoya[1,a)] Yuuki Takano[1,b)] Atsuko Miyaji[1,c)]

Abstract: In a society where many devices such as smartphones and personal computers are connected to the Internet, it is desirable that network functions that connect multiple devices be implemented more safely. In fact, in network functions such as network switches and routers, packets may not reach their destinations due to a run-time bug where packets are dropped due to model violations. By verifying the network function, it can be guaranteed that the network function of the actual network follows the model faithfully. F* is a functional language for program verification. In this paper, the management of the MAC address of the switching hub, which is one of the network functions, was realized using the F*. As a result, the filtering function was verified, and a network function with no error in specifications was realized.

Keywords: network function, F*, program verification

1. はじめに

1.1 研究背景

スマートフォンやパソコンといった多くの機器がインターネットに繋がる社会において，複数の装置同士をつなげるネットワークファンクション (以下，NF) はより安全に実装されることが望まれる．しかし，実際に人によって書かれたプログラムは何かの誤りで仕様とは異なる動作をすることがあり，人間による確認だけで正しい動作を保証することは困難である．実際に Hongqiang Harry Liu らは Microsoft のサービスにおいてソフトウェアのバグが大きな問題を引き起こしていることを指摘している [1]．この問題を解決するために様々な手法で NF を検証する研究がなされているが，複雑な手法を用いたり一部の検証にとどまっているのが現状である．

そのため NF の検証を実用的なものするにはより扱いやすい手法を確立する必要がある．したがって，複雑な型システムをもつことでプログラムの検証を行うことのできる F*言語を用いることで，従来とは異なる NF の検証手法を模索した．

1 大阪大学
Osaka University
a) hosoya@cy2sec.comm.eng.osaka-u.ac.jp
b) ytakano@cy2sec.comm.eng.osaka-u.ac.jp
c) miyaji@comm.eng.osaka-u.ac.jp

ⓒ 2020 Information Processing Society of Japan

1.2 本研究の目的

現在までに公表されてる様々な研究において，様々な手法を用いて NF を検証する技術が考えられている．しかし，仕様の一部を検証するものであったり，仕様を完全に検証できるものであっても複雑な手法を用いる必要があるため実用的であるとは言い難い．本研究では F*言語を用いることで複雑な手法を用いることなく NF の検証を行うことを目的としている．本手法は既存の手法と同様に C 言語での実装を可能としている反面，F*言語を変換するだけで検証が完了するため手順としてはシンプルなものとなっている．しかし，F*言語の型システムを用いて NF の仕様を表現するには専門的な知識が必要であるため，依然として実用性には問題がある．本手法に則って，F*言語を用いることで MAC アドレスを管理したスイッチングハブは仕様に誤りなく動作する．

1.3 本論文の構成

本論文の構成は以下のとおりである． 2 章では本論文に関する既存研究である A Formally Verified NAT [2] について記載する． 3 章では F*言語を用いた NF の検証の設計原理について記載する． 4 章では 3 章の設計原理に基づいて F*言語を用いた NF の検証方法を提案する． 5 章では実際にスイッチングハブにおいて F*言語を用いることで MAC アドレスの管理を行い，仕様に誤りなく実装する手法について記載する． 6 章では本検証手法について定性的，定量的に評価をする． 7 章では本研究のまとめを行い，今後の課題について記載する．

2. 既存研究

Arseniy Zaostrovnykh らは Lazy Proofs という証明技術に則り，自作の Vigor と呼ばれるツールチェーンを用いることで，C 言語の NAT のソフトウェアを検証する手法を提案した [2]．Vigor は C 言語で構築された NF のソフトウェアをステートフルな部分とステートレスな部分に分けて証明を行う．ステートレスな部分自体に関してはシンボリック実行によって証明を行うことができるが，ステートフルな部分に関しては容易に証明を行うことができない．よって，この論文ではあらかじめ手動で証明を行ったライブラリを参照することでステートフルな部分を証明する．また証明は全体としてステートレスな部分を主軸として行われ，完全にステートフルな部分を省くことはできないため図 1 の様にシンボリックモデルで置換することで証明を行う．以上の様に複数の証明が存在するため，Lazy Proofs という技術を用いることで効率よく証明を行う．

3. 設計原理

本章では 2 章で記述した NAT の検証とは異なったアプローチで，NF の検証を行う方法について説明する．3.1 節では NF が動作する際に考慮されるべき条件について，3.2 節では前節の条件をふまえて NF を検証する手法の設計原理を記載する．

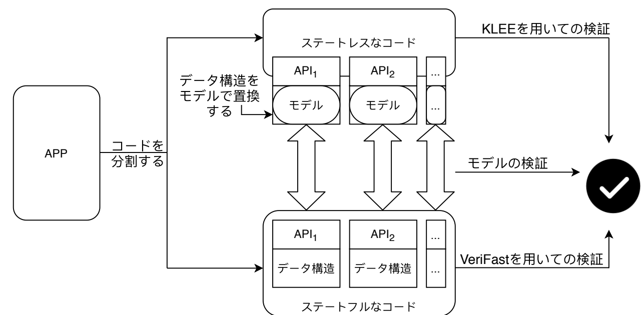

図 1 Vigor のワークフロー

3.1 満たされるべき条件

現在，世の中には様々な NF があふれているが，それらはバグや脆弱性といった問題によって本来意図されない動作を行ってしまう可能性を含んでいる．そこで NF を設計する上で考慮すべき条件は以下が挙げられる．

- ソフトウェアがクラッシュしない
- ソフトウェアが仕様に従う
- 全てのパケットが最終的に処理される
- 適切な言語が採用される

3.1.1 クラッシュの回避，仕様への従属，全パケットの処理

様々な機器がインターネットにつながる IoT 社会において，NF のソフトウェアが正確に動作しないことは大きな問題となりうる．自動運転をしている車との情報のやりとりにおいて，誤った情報が送られたり，適切な情報が与えられないことによって交通事故が発生する可能性が考えられる．また医療の場面においては，遠隔操作によって手術をする際などに機械の誤った動作につながる可能性も考えられる．したがって，今後，より生活に身近になってくる NF が正確に動作をすることは重要である．

3.1.2 適切な言語の採用

今日，世の中には様々なプログラミング言語があふれているが，それぞれ異なった特性をもっており，どれほど普及しているのかや歴史の浅い深いも様々である．ここで，採用する言語に必要な条件の一例を考えてみる．一般的に使われている NF は 1 種類に限らず複数存在するため，ある NF にだけ高いパフォーマンスを発揮できる言語を採用することは望ましいとはいえない．どの言語を採用するかを考えるにあたって，例で挙げたような条件は複数存在する．したがって，いくつかの条件をもとに採用する言語を選択する必要がある．

3.2 NF を検証する手法

3.1 節を満たすような NF の検証に関する研究は行われているが，一部を満たすだけにとどまっていたり，複雑な手法で証明をおこなったりと専門の知識がない人にとってはまだまだ扱いにくいのが現状である．そこで本論文で提案する，よりシンプルな検証の手法についての設計原理は以下のとおりである．

- 適切な言語を用いた NF の構成
- プログラム検証言語による NF の検証

3.2.1 適切な言語を用いた NF の構成

NF には様々な種類が存在するため，対応する機器の範囲が広く柔軟に対応できる言語を採用する必要がある．また，効率的に NF を実装するため，できるだけネットワーク技術を扱うのに特化した既存のライブラリをもつ言語を採用したい．加えて，NF において処理の速さは重要な要素となってくるため，完成したプログラムが高速で動作することも必要となってくる．

3.2.2 プログラム検証言語による NF の検証

ソフトウェアが NF の仕様に従い，かつ安全に動作することを示すためにはソフトウェアの動作を機械的に保証する必要がある．プログラミング言語には言語ごとに様々な型が存在し，int のような整数を示すだけのものもあれば，依存型のような型の中に値が含まれる型も存在する．型検査の際，これらの型はその言語の仕様にしたがって正しく宣言されているのか確認され，検査を通った場合のみ実行することができる．つまり型が複雑になればなるほど，プログラムが複雑な仕様を満たすことを型検査で保証することができるといえる．したがって，NF を検証するためには複雑な型を定義することのできる言語を用いてソフトウェアを構築し，動作を機械的に保証することが必要となってくる．

4. 設計

4.1 設計概要

本章ではプログラム検証言語を用いた NF の設計概要について説明する．図 2 に示すように，本 NF は F*言語をもちいてソフトウェアを構築し，型検査によって仕様に則って安全に動作することを保証したのちに，骨組みとなる C 言語のコードから呼び出せるようにすることで実現される．本論文内では，NF の一つであるスイッチングハブにおいて最も基礎的な，適切なポートにパケットを転送するために MAC アドレスを管理する部分を F*言語で構築し，スイッチングハブの C 言語のコードに組み込んで実装するところまでを行う．その他の設計については本論文の今後の課題とする．

4.2 C 言語を用いた NF の構成

C 言語は，DPDK [3]，XDP [4]，Socket といったパケットを扱ううえで豊富で充実したライブラリを備えており，コンパイラによって機械語に変換され実行されるため高速に動作することができる．それゆえ，C 言語を用いることでパケットを速く処理することの求められる NF の実装においては，高いパフォーマンスを実現できる．しかし，C 言語は複雑な型システムをもたないため，プログラムの検証に適しているとは言い難い．そのため，本論文では，検証に適した F*言語で MAC アドレスの管理に関するプログラムを構築し，検証したのちに C 言語から呼び出せるように変換することによって C 言語での実装を実現する．また，図 2 に示すように最終的な目標としては NF を可能な限り F*言語で構築し C 言語に変換した後，枠組みとなる C 言語に組み込んで実装することで，パフォーマンスと安全性の両立を実現する．

4.2.1 F*言語を用いた関数やデータ構造の検証

昨今，NF の検証に関して様々な研究がなされているが，概ねパケットの到達可能性や無限ループに陥らないことを証明するものである．一方で，NF としての仕様を満たし安全に動作することを検証する手法も研究されている．しかし，実装に適している C 言語のコードを直接検証したいがゆえにコードを属性ごとに分け，それぞれ異なる方法で証明を作成し，その証明を組み合わせるといった複雑な手法になっている．

本論文では，NF を検証するのに F*言語を用いる．F*言語は依存型，モナド効果，リファインメント型の要素が組み込まれた複雑な型システムをもつため，NF の仕様や動作の安全性を型で表現することができる．また，F*言語で書かれたプログラムはコンパイル時に行われる型検査を通れば型に沿った動作をすることが保証されるため，NF の検証を行うことも可能である．このとき，プログラムの検証には Z3 Theorem Prover [5] と手動証明が用いられ，検証のためにプログラムを分ける必要が生じない．したがって，本論文ではスイッチングハブの MAC アドレスを管理する部分を F*言語の型を用いて表現することで，仕様的に誤りのないスイッチングハブを実現する．

5. 実装

5.1 内部構成・変換手法

4 章で記載したとおり，スイッチングハブの実装の大部分は C 言語を用いて，MAC アドレスの管理については F*言語を用いて行った．5.1 節では C 言語のプログラムの内部構造，5.2 節では F*言語のプログラムの内部構造，5.3 節では F*言語を C 言語で呼び出す方法について説明する．

5.1.1 C 言語

スイッチングハブを実装するためには，ポートの割り当てを管理するためにパケットに含まれる MAC アドレスの情報を得る必要がある．よって，まず C 言語で扱うことのできる Socket の一つである Raw Socket を用いること

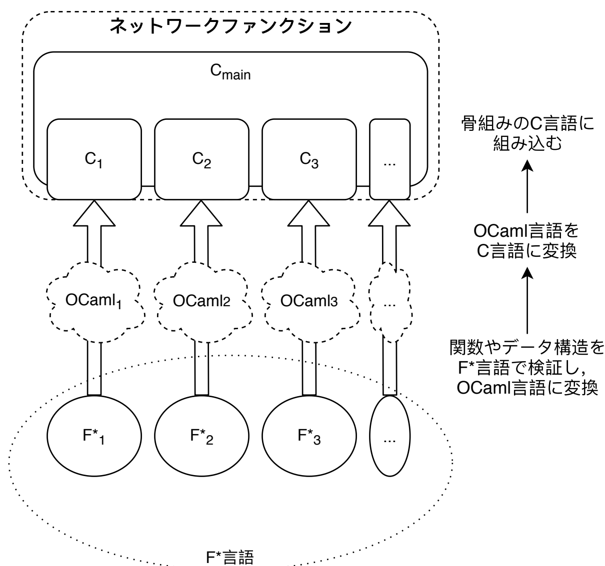

図 2　ネットワークファンクションの設計

で MAC アドレスの情報の含まれた生のパケットを入手する．このとき Raw Socket をプロミスキャスモードで使用することで，送られてくる全てのパケットを得ることができる．一般的に，ある端末が通信を開始する際には繋がっている端末全てに一斉に ARP 要求が送られ，目的の端末から ARP 応答が返ってきた場合は ARP テーブルを更新し，以降は IP プロトコルを用いて通信を行う．この一連の通信方法において，パケットのカプセル化に IP や ARP といった異なったネットワーク層のプロトコルが扱われる．しかし，MAC アドレスはその下層であるネットワークインターフェース層で扱われるため，ネットワーク層のプロトコルに関係なくイーサネットヘッダの中から取得することができる．よって，Raw Socket で入手したパケットのイーサネットヘッダから送信元 MAC アドレスと宛先 MAC アドレスを入手することができる．次に，F*言語を用いて検証された関数を呼び出すことで，パケットに格納される送信元 MAC アドレスの判定を行う．MAC アドレスが意図するものであった場合，関数は Bool 型で true を返し，そうでない場合 false を返す．これをもとに Raw Socket で入手したパケットを適切なポートに転送するか破棄するかを判断し，実行する．

5.1.2　F*言語

F*言語上では MAC アドレスを引数にとり，その MAC アドレスが適当かどうかの判定を Bool 型で返す関数を実装した．F*言語は型システムにリファインメント型を用いることができる．よってソースコード 1 に示すように関数を定義することで，コンパイルする際に引数が MAC アドレスを表す整数であり，関数内に存在する MAC アドレスのデータと照合した結果返される値が Bool 型であることを保証する．これによって意図する MAC アドレスをもつ端末からのパケットを適切に転送することを可能にする．

5.1.3　C 言語からの F*言語の呼び出し

F*言語で実装された関数を C 言語で呼び出す方法について説明する．ソースコード 1 の F*言語の関数は，make コマンドを使うことで Makafile ファイルの指示に従ってコンパイルされる．これによって F*言語の関数は型検査された後，ソースコード 2 で示すように OCaml 言語の関数として出力される．OCaml 言語ではコールバック関数を用いることで関数を C 言語から呼び出すことが可能となるが，現段階では呼び出すための OCaml 言語の関数を手動で記述する必要がある．このときソースコード 2 の let _ は呼び出すための関数を手動で定義している部分である．

次に，C 言語から呼び出すために OCaml 言語の関数を C 言語のオブジェクトファイルにコンパイルする．このとき様々なライブラリを参照する必要が生じるが，OCaml 言語の標準的な仕様では unix ライブラリしか明示されていない．また，それだけの参照ではコンパイル時にエラーが発生するためその他のライブラリに関して自ら探す必要が生じる．同様に，オブジェクトファイルを C 言語と合わせてコンパイルする際にも，複数のライブラリを参照する必要が生じる．一方，C 言語では main 関数内で OCaml コードを初期化するための caml_startup(argv); を宣言する必要がある．以上の作業を行うことで，ソースコード 3 に示すような F*言語を呼び出す関数が実装されたプログラムが出力される．

しかし，この方法ではプログラムを実行した際にセグメンテーション違反が生じ，実際に F*言語の関数を呼ぶことができない．その原因と考えられるのが，C 言語のオブジェクトファイルにコンパイルする際に用いる libzarith.a である．現状，このライブラリを参照してコンパイルを行うと F*関数の呼び出しの際にセグメンテーション違反が生じてしまう．しかし，F*言語から作られた C 言語のオブジェクトファイルをコンパイルするにはこのライブラリの参照が不可欠である．よって，F*言語を変換することで得られた OCaml 言語の内容を書き換えることで，このライブラリの参照の必要なくコンパイルする方法を考えた．大前提として，内容を書き換えると検証の意味がなくなるため，以下では手動で F*言語の型を OCaml 言語の型に変換する操作のみを行った．

ソースコード 2 における Prims は F*言語に存在する特有のライブラリであり，OCaml 言語上で F*言語の型を扱うためのものであると考えられる．実際に，この Prims ライブラリを扱うために C 言語のコンパイル時に zarith.a が必要となるため，このライブラリを用いずに型を表現する必要が生じる．したがって，ソースコード 4 に示すように F*言語の型を表していた部分を OCaml 言語の型に書き換えた．これによって，プログラムが F*言語のライブラリを参照する必要がなくなったため，コンパイルが可能となった．その結果，F*言語の関数を呼び出すことのできる C 言

ⓒ 2020 Information Processing Society of Japan

ソースコード 1 F*言語で実装した関数

```
1  module Macaddress
2
3  let canValidate (f:int) =
4       match f with
5       | 0x0800278a4e3f -> true
6       | _ -> false
```

ソースコード 2 F*言語の関数を OCaml 言語に出力した結果

```
1  open Prims
2  let (canValidate : Prims.int -> Prims.bool) =
3    fun f ->
4      match f with
5      | _133 when _133 = (Prims.parse_int "0x0800278a4e3f") -> true
6      | uu____135 -> false
7
8  let _ = Callback.register "Verifyingmac" canValidate
```

表 1 本研究と既存研究の特性比較

	既存研究 (Vigor)	本研究 (F*)
C 言語での実装	○	○
手法のシンプルさ	×	○
専門知識の必要性	△	△

語のプログラムを実装することができた．

以上の様に簡単な F*言語の関数であっても C 言語に変換するためには複数のライブラリを参照する必要があり，中には参照する必要があるにも関わらずプログラム実行に悪影響を及ぼすライブラリも存在する．したがって，今後 F*言語を用いて NF を検証するためには，各ライブラリを修正するか，コンパイル環境を整理する必要がある．

6. 評価

6.1 定性的評価

本研究の強みの主たる部分は，様々な手法を組み合わせることなく既存のツールを用いることで，シンプルに NF の検証を行うことができる点である．ここで，表 1 に既存研究と本研究の特性の比較について示す．

表 1 のように，本研究と A Formally Verified NAT とを比較すると最終的に C 言語を用いて NF を実装する点では同じである．一方で，本研究は骨組みとなる C 言語の構成，F*言語の構成，呼び出すための F*言語のコンパイル，の大まかに 3 つのステップによって検証を実現している．そのため既存研究のように検証されたライブラリを用意し様々な証明をする必要はなく，F*言語で関数を構成しコンパイルが成功することで NF を検証できるため，本研究の方がよりシンプルであるといえる．また，既存研究では検証のために用いるツールやライブラリを専門的な知識を用いて作成する必要がある．本研究においては既存の F*言語を用いるためそのような心配はないが，F*言語の型システムを用いて NF の仕様を適切に表現したり，正しくコンパイルするためにライブラリを参照する必要がある．したがって，どちらも専門知識なしに実装することが難しい．

6.2 機能評価

本節では 5 章で記載したスイッチンハブを用いたパケットの転送について実行結果を示す．まず，図 3 に示すようなパケット転送を行ったネットワークの環境について説明する．ノード 1 の eth1 とノード 2 の eth1 は同一のネットワーク上に存在し，同様にノード 2 の eth2 とノード 3 の eth1 も同じネットワーク上に存在する．ただし，これらのネットワークは別物であるため，ノード 1 とノード 3 が直接通信をすることはできない．ノード 2 において本スイッチングハブを実装することでノード 1 とノード 3 間の通信が可能となる．

実際に，ソースコード 5 に示すように ping コマンドを用いてパケットの送信者であるノード 1 からノード 3 に対するパケットを送信した．このとき 1 行目は ping コマンドであり，172.16.0.13 はノード 3 の IP アドレスを表している．2 行目以降では 3 回の ping 要求をノード 3 の IP アドレスに送り，全てに対して ping 応答が返ってきたこととその返答速度を表している．

また，ソースコード 5 を実行した際のノード 2 の動作をソースコード 6 に示す．1 行目ではコマンドライン引数によって Raw Socket を用意するネットワークインターフェースを指定している．次に 2, 3 行目ではノード 2 の eth1 に対しては sockA という名前の Raw Socket を，eth2 に対しては sockB という名前の Raw Socket を作成したことを表している．このとき，index はそれぞれの socket に対して振り当てられた番号であり，sockA には 3 が sockB には 4 が割り当てられている．

ⓒ 2020 Information Processing Society of Japan

ソースコード 3 C 言語における F*言語の関数の呼び出し

```
1   bool verifyingmac(long n){
2       static const value *mac_closure = NULL;
3       if(mac_closure == NULL)
4             mac_closure = caml_named_value("verifyingmac");
5       return Bool_val(caml_callback(*mac_closure, Val_long(n)));
6   }
```

ソースコード 4 手動型変換をおこなった OCaml 言語の関数

```
1   let (canValidate : int -> bool) =
2     fun f ->
3       match f with
4       | _133 when _133 = (0x0800278a4e3f) -> true
5       | uu____135 -> false
6
7   let _ = Callback.register "Verifyingmac" canValidate
```

次に 5〜12, 14〜21, 23〜30, 32〜39, 41〜48, 50〜57 行目では片方の Raw Socket で受信したパケットがもう片方の Raw Socket に送られたことを表している．5〜12 行目について詳細にみてみると，5, 6, 9 行目では SockA (3) からの 98 バイトのパケットを SockB (4) に転送したことを表している．10, 11 行目では転送したパケットの送信元 MAC アドレスと宛先 MAC アドレスが出力されているため，転送されたパケットがノード 1 の eth1 からノード 3 の eth1 へ宛てたものであることを表している．12 行目ではパケットのインターネット層で扱うプロトコルが IPv4 であることを表している．また，14〜21 行目では 5〜12 行目のパケットに対する応答のパケットを表している．以降についても同様に転送を行ったパケットの情報が出力されており，ノード 2 を介してノート 1 とノード 3 が通信を行っていることがわかる．

次にソースコード 7 では， 5 章のソースコード 1 の 5 行目の MAC アドレスを “0x000000000000” に変更した場合，つまりノード 1 からのパケットは転送しない場合の動作を示す．このとき 1 行目は上記と同様に 172.16.0.13 へ ping を送信するコマンドを表している．しかし，2 行目以降は 3 回の ping 要求全てに対して ping 応答は返ってこず，パケットが破棄されたことを表している．また，ソースコード 7 を実行した際のノード 2 の動作をソースコード 8 に示す．1〜3 行目はソースコード 5 と同様にノード 2 の eth1 と eth2 に対して SockA と SockB が用意されることを表している．しかし，4 行目以降の出力はなくノード 2 を経由してパケットが転送されていないことがわかる．

したがってソースコード 5, 6 と 7, 8 の比較により，F* 言語の関数で MAC アドレスを管理することで，実際に仕様に誤りなくパケットを転送することが可能になったといえる．

7. まとめ

本論文では，F*言語を用いることで実現される，検証可能な NF の設計と実装について記載した． 5 章で記載したとおり，スイッチングハブの MAC アドレスを管理する部分を F*言語の関数で実装し，OCaml 言語に変換した後 C 言語のオブジェクトファイルにコンパイルし，スイッチングハブの C 言語から呼び出すことができた．この手法を用いることで，複雑な手順を要さずに NF の検証を行うことが可能となった．

今後の課題として，F*言語を用いた NF と C 言語で実装された NF とのパケットの処理速度の比較を行い評価することが挙げられる．また安定して F*言語を C 言語から呼び出せるようにするために，コンパイル環境を整理するかライブラリを修正することが挙げられる．最後に，本研究は研究段階にある検証を目的とした F*言語の実用化を助長し，安全な NF の普及を促進するものであるため，有意義な研究であると考えられる．

謝辞

本研究の一部は科学技術振興機構（JST）の CREST (JPMJCR1404) 及び文部科学省「Society5.0 に対応した高度技術人材育成事業成長分野を支える情報技術人材の育成拠点の形成 (enPiT)」さらに文部科学省の平成 30 年度「Society 5.0 実現化研究拠点支援事業」の助成を受けています．

参考文献

[1] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles,*

ⓒ 2020 Information Processing Society of Japan

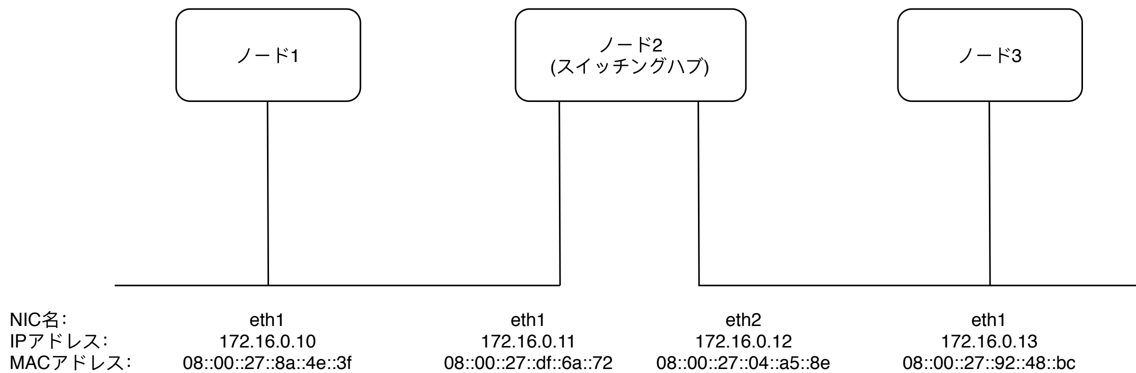

図 3 実行ネットワーク環境

ソースコード 5 ノード 1 における ping コマンド

```
1  ping 172.16.0.13
2  PING 172.16.0.13 (172.16.0.13) 56(84) bytes
       of data.
3  64 bytes from 172.16.0.13: icmp_seq=1 ttl
       =64 time=0.816 ms
4  64 bytes from 172.16.0.13: icmp_seq=2 ttl
       =64 time=0.747 ms
5  64 bytes from 172.16.0.13: icmp_seq=3 ttl
       =64 time=0.563 ms
6
7  --- 172.16.0.13 ping statistics ---
8  3 packets transmitted, 3 received, 0%
       packet loss, time 32ms
9  rtt min/avg/max/mdev =
       0.563/0.708/0.816/0.111 ms
```

Shanghai, China, October 28-31, 2017, pp. 599–613. ACM, 2017.

[2] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina J. Argyraki, and George Candea. A Formally Verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pp. 141–154. ACM, 2017.

[3] Linux Foundation. Data Plane Development Kit, 2020. https://www.dpdk.org/.

[4] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress data path: fast programmable packet processing in the operating system kernel. In Xenofontas A. Dimitropoulos, Alberto Dainotti, Laurent Vanbever, and Theophilus Benson, editors, *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*, pp. 54–66. ACM, 2018.

[5] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, Vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340. Springer, 2008.

ソースコード 6 ノード 1 における ping コマンド

```
1   sudo ./SWITCHINGHUB eth1 eth2
2   Interface1 (SockA) : eth1 (index 3)
3   Interface2 (SockB) : eth2 (index 4)
4
5   forward 98 bytes to IF4
6   FromInterface = 3
7   PacketType = 3
8   HeaderType = 1
9   Frame Size = 98
10  Ether SrcAdr = 08:00:27:8a:4e:3f
11  Ether DstAdr = 08:00:27:92:48:bc
12  Ether Type = 800
13
14  forward 98 bytes to IF3
15  FromInterface = 4
16  PacketType = 3
17  HeaderType = 1
18  Frame Size = 98
19  Ether SrcAdr = 08:00:27:92:48:bc
20  Ether DstAdr = 08:00:27:8a:4e:3f
21  Ether Type = 800
22
23  forward 98 bytes to IF4
24  FromInterface = 3
25  PacketType = 3
26  HeaderType = 1
27  Frame Size = 98
28  Ether SrcAdr = 08:00:27:8a:4e:3f
29  Ether DstAdr = 08:00:27:92:48:bc
30  Ether Type = 800
31
32  forward 98 bytes to IF3
33  FromInterface = 4
34  PacketType = 3
35  HeaderType = 1
36  Frame Size = 98
37  Ether SrcAdr = 08:00:27:92:48:bc
38  Ether DstAdr = 08:00:27:8a:4e:3f
39  Ether Type = 800
40
41  forward 98 bytes to IF4
42  FromInterface = 3
43  PacketType = 3
44  HeaderType = 1
45  Frame Size = 98
46  Ether SrcAdr = 08:00:27:8a:4e:3f
47  Ether DstAdr = 08:00:27:92:48:bc
48  Ether Type = 800
49
50  forward 98 bytes to IF3
51  FromInterface = 4
52  PacketType = 3
53  HeaderType = 1
54  Frame Size = 98
55  Ether SrcAdr = 08:00:27:92:48:bc
56  Ether DstAdr = 08:00:27:8a:4e:3f
57  Ether Type = 800
```

ソースコード 7 ノード 1 における ping コマンド

```
1   ping 172.16.0.13
2   PING 172.16.0.13 (172.16.0.13) 56(84) bytes
        of data.
3
4   --- 172.16.0.13 ping statistics ---
5   3 packets transmitted, 0 received, 100%
        packet loss, time 44ms
```

ソースコード 8 ノード 1 における ping コマンド

```
1   sudo ./SWITCHINGHUB eth1 eth2
2   Interface1 (SockA) : eth1 (index 3)
3   Interface2 (SockB) : eth2 (index 4)
```

ⓒ 2020 Information Processing Society of Japan