Vol.2022-DPS-190 No.15
Vol.2022-CSEC-96 No.15
2022/3/10

Rust言語を用いたNATの実装と検証

細谷 昂平[1,a)]　高野 祐輝[1,b)]　宮地 充子[1,2,c)]

概要：SDN や NFV といった技術の発展によってソフトウェアでのパケット処理の需要が高まっている．10-100GbE の普及により端末間のデータ通信は高速に行える環境が整備されつつあるが，ソフトウェアでの処理はハードウェアに比べて低速であるため通信のボトルネックとなっている．この問題を解決する一例として DPDK を代表とするカーネルバイパス技術が挙げられる．また，ネットワークサービスにおける深刻な障害の要因としてソフトウェアのバグが占める割合が高いという現状があるため，ネットワークファンクションを構築する上でソフトウェアのメモリ安全性や仕様の検証を行うことが望まれる．本論文では，上述の DPDK を用いてユーザー空間での高速なパケット処理を実現し，Rust 言語を用いてマップを構成することで仕様に誤りなく動作する NAT を実装する．Rust 言語で構成されるマップは Rust 言語の検証器である Prusti を用いて検証され，追加，削除，参照といった操作をメモリ安全かつ仕様に誤りなく行うことができ，本手法を用いることで他のネットワークファンクションにおいてもデータ構造の検証が可能となると考えられる．

キーワード：Rust，Prusti，DPDK，NAT，プログラム検証

Implementation and Verification of A NAT in Rust

Abstract: The development of technologies such as SDN and NFV has increased the demand for software packet processing. With the spread of 10-100GbE, a high-speed environment for data communication between terminals is being developed, but software processing is slower than hardware, and thus becomes a bottleneck for communication. Kernel bypass technology, such as DPDK, is an example of a solution to this problem. In addition, software bugs account for a large percentage of serious failures in network services, so it is desirable to verify the memory safety and specifications of software when constructing network functions. In this paper, we implement a NAT that can operate without errors in the specifications by realizing fast packet processing in user space using the DPDK described above and constructing maps using the Rust language. The map constructed in the Rust language is verified using Prusti, a verifier for the Rust language, and operations such as addition, deletion, and reference can be performed memory-safely and without specification errors.

Keywords: Rust，Prusti，DPDK，NAT，program verification

1. はじめに

1.1 研究の背景

SDN や NFV といった技術の発展によってソフトウェアによるパケットの処理の需要が高まっている．Linux などの OS におけるソフトウェアでのパケットの処理は，受信パケットがカーネル空間のネットワークスタックで複数の処理を実施された後に，ユーザー空間に送られることで初めて可能となる．このため，10-100GbE の普及など，端末間の高速な通信環境の設備にも関わらず，カーネル空間でのパケット処理がボトルネックとなるのが現状である．上記問題を解決するためにカーネルバイパス技術が存在し，DPDK[1] や AF_XDP[2] などが代表に挙げられるが，AF_XDP においては未だ開発段階であり，異なる NIC 間でのパケット転送を同時に双方向で実装することが難しい

[1] 大阪大学大学院 工学研究科
Graduate School of Engineering, Osaka University
[2] 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology(JAIST)
[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

ため，DPDK を用いることが望まれる．

一方，Hongqiang Harry Liu らによってネットワークサービスの深刻な障害とソフトウェアのバグは密接に関わっていることが報告されている [3]．ネットワークファンクション (NF) におけるソフトウェアのバグの未然防止を目指し，メモリ安全性と仕様の検証について研究がされてきた．しかし，Aragog[4] のようにメモリ安全性と NF 仕様の両方の検証ができない場合や，Vigor[5] のようにツール自体が複雑な場合や，ツールを使うための専門的な知識が必要な場合など利用場面が限られるのが現状である．また，Rust 言語は C 言語に匹敵する実行速度をもちながら，メモリ安全にメモリ管理を行えるため注目を集めており，Prusti[6] といった簡素な表現を用いて形式的に関数の検証を行えるツールも開発されている．

1.2 研究の目的

現在までに NF におけるソフトウェアのバグを検証するために様々なツールが提案されてきたが，ツールを使うための専門的な知識を必要とする場合やツール自体が複雑で扱いが難しい場合，汎用的な利用に適さない場合など利用者や利用場面が限られるのが現状である．

本研究では，ソフトウェアにおける高速なパケット処理のために DPDK，メモリ安全性の実現のために Rust 言語，Rust 言語の関数の振る舞いを検証するために Prusti を用いることで，専門的な検証技術を用いずに簡易に検証を行う手法を設計する．さらに，上記設計に基づき，例として NF のうちデータ構造を用いて動作する NAT を実装することで，設計の実証を目的とする．

1.3 本論文の構成

本論文の構成は，以下の通りである．2 章では，本研究の基となるネットワーク検証に関する既存研究について述べる．3 章では，提案 NF の設計概要と例として実装する NAT の検証すべき条件について考察する．4 章では，本 NF の設計に基づいた NAT の具体的な実装とその検証手法について述べる．5 章では，本 NAT の定性的，機能，パフォーマンス評価について考察する．6 章では，本研究のまとめと今後の課題について述べる．

2. 既存研究

Arseniy Zaostrovnykh らは，Vigor という独自のツールチェーンを用いて，C 言語で書かれた NF がクラッシュせず，メモリ安全に動作し，また Python で書かれた NF が満たすべき仕様に則って動作することを検証する手法を提案した [5]．Vigor は，主に VerFast proof checker[7]，Modified KLEE symbolic execution engine[8]，Validator といったツールを組み合わせ，Lazy Proofs という検証を組み合わせる技術を用いることで上記の検証を可能にしている．具体的には，NAT，ロードバランサー (Maglev)，MAC 学習ブリッジ，ステートフルファイアウォール，トラフィックポリサーについて上記の条件を満たしながら動作することを検証することに成功した．

Nofel Yaseen らは，Aragog という NF の実行時検証システムを提案した [4]．Aragog は，分散型，ステートフル，時間推移といった様々な NF に対して検証を行うことができ，さらに実行時に検証が行えるという特徴を持つシステムである．また，独自の言語を用いることで違反仕様を簡潔に記述することを可能としており，グローバルな検証システムとローカルな監視装置を組み合わせることにより検証を行う．ローカルな監視装置はネットーワークファンクションの状態遷移を定義することによって，遷移ごとに検証プロセスの要否を判断し，必要なものだけがグローバル検証システムによって検証されるため，余分な検証を省略することができ，さらにその処理を利用可能なグローバルとローカルの構成要素に分散して実行することにより，パフォーマンスの向上を実現した．

Hongqiang Harry Liu らは，Microsoft のネットワークサービスで深刻な障害を引き起こした原因として，ソフトウェアによるバグが 1/3 以上を占め，例としてルーターなどのミドルボックスによるバグが生じていることを報告した [3]．この問題を解決するために，CrystalNet というクラウド規模で高性能に機能するネットワークエミュレータを提案し，コンテナや仮想マシンのネットワークに本番の環境を構築した上で，実際のネットワークデバイスのファームウェアを動作させることで，擬似的にテストを行うことを可能にした．

我々は，Rust 言語によるフィルタ機能を付加したソフトウェアブリッジの実装と検証 [9] において，本論文の基となる Rust 言語，Prusti，DPDK を用いた NF の設計と Rust 言語と DPDK を用いたソフトウェアブリッジの実装を行い，その際に利用するマップを検証するために必要な条件を考察した．

3. 設計原理，設計

本研究では，我々の提案した NF[9] の設計原理と設計に従って NAT の実装を行う．

3.1 設計概要

本 NF は，図 1 に示されるように Rust 言語で実装することでメモリ安全性を保証された関数が検証器により仕様に従うことを検証された後に，C 言語で構築されたユーザー空間での高速なパケット処理を可能にする DPDK フレームワークに組み込まれることで実現される．従って，DPDK フレームワークの正しさは提供元に依存することとなる．

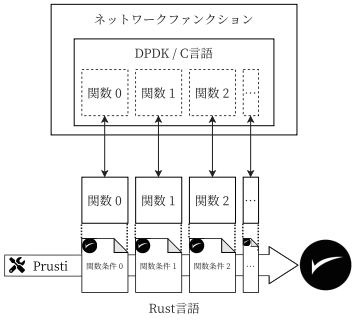

図 1 本 NF の設計概要

本論文では，数ある NF のうち NAT について設計と実装を行う．NAT の実装は，同様のデータ構造やパケットの変換処理を用いる他の NF への拡張が可能であり，検証の基盤を確立する上で最適であると考えられる．また，その他の NF の実装とパケット通信の検証については本論文の今後の課題とする．

3.2 NAT の設計

本節では，本研究で実装を行う NAT についての構成とその検証要件について記述する．本 NAT はグローバルネットワークとプライベートネットワークのパケットをお互いに変換するための機能を有する必要があるため，2 つの NIC を持つマシン 1 台から構成され，マシンは汎用性を考慮して LinuxOS を搭載するものとする．本 NAT の概要は，図 2 に示される通りである．

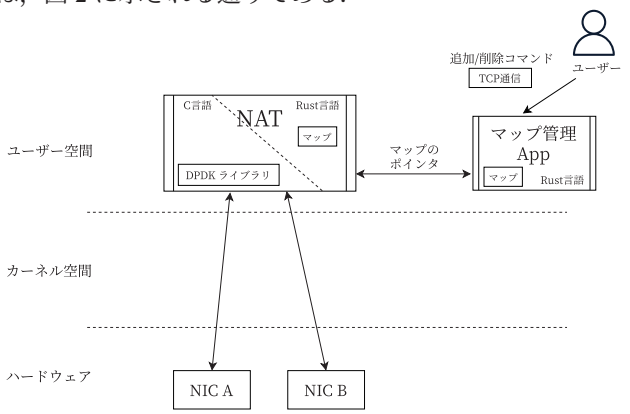

図 2 本 NAT の設計概要

本 NAT は IP アドレスとポート番号の情報をグローバルとプライベートで対応づけてマップに格納することで変換情報を記憶し，そのマップが TCP 通信によって管理される機構を Rust 言語で構築した後，それらを C 言語で構成された DPDK フレームワークから呼び出すことによって実装される．

3.3 Prusti によるマップの検証

本 NAT では，IP アドレスとポート番号の変換の肝となるマップについて満たすべき仕様に準拠するか検証することを目的とする．本 NAT で使用するリストマップは Rust 言語により構成されるため，その検証器である Prusti を用いての検証が可能である．一般的にマップは追加，削除，参照という操作が行われることを前提として構成されており，本リストマップに関しても上記の 3 つの操作が行われることとする．まず，これらの操作が正しく行われることを保証するには次の 3 つの条件を満たす必要がある．

- 空のマップには何も格納されていない
- 追加された key と value が実際に追加されている
- 削除された key と value が実際に削除されている

さらに，NAT はグローバルネットワークとプライベートネットワークの情報を対応づけて管理する必要があるため，図 3 に示すように 2 つのマップを互いに逆方向に活用する必要があり，上記に加えて次の条件も満たす必要がある．

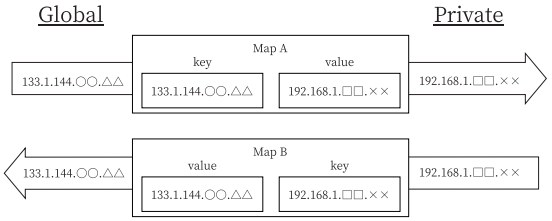

図 3 NAT におけるマップの活用イメージ

- ネットワークの対応情報が追加された際に，実際に双方向で追加されている
- ネットワークの対応情報が削除された際に，実際に双方向で削除されている

従って，本 NAT ではリストマップについて次の条件が満たされることを検証する．

- 空のマップには何も格納されていない
- ネットワークの対応情報が追加された際に，実際に双方向で追加されている
- ネットワークの対応情報が削除された際に，実際に双方向で削除されている

4. 実装

本章では，3 章で提案した設計をもとに本 NAT の実装について説明する．まず，4.1 節で C 言語で構成された DPDK プログラムから Rust 言語の関数を呼び出す手法を紹介し，4.2 節で Rust 言語の関数が 3.3 項で定義した条件を満たすことを検証する手法について示した後，4.3 節で本 NAT の機能について記載する．

4.1 C 言語プログラムにおける Rust 言語関数の呼び出し

本節では，DPDK 上から Rust 言語を呼び出す際に必要な手順について述べる．まず，Rust 言語に標準で実装されているライブラリの出力機能と呼び出し用の関数定義を用いてプログラムを構築する．その後ヘッダファイルに関しては，cbindgen[10] という Rust のプログラムを用いる

ことで準備することができるが，配列のアドレスなど特定の型を扱う場合に変換が必要であったり，サポートされていない型も存在するため注意が必要である．以上の準備を行った後，C 言語で構築された DPDK のコードをコンパルする際に，ソースコード 1 の 2 行目のようにライブラリ名とライブラリ自体の存在する階層を指定することによって Rust 言語の呼び出しが可能となり，5 行目に示すように，ライブラリのパスと必要な引数を渡すことによって起動が可能になる．

ソースコード 1 C 言語への Rust 言語関数の組み込み

```
1  //C 言語への Rust 言語関数の組み込み
2  gcc -g（組み込み先のC 言語).c -l(ライブラリ
       名) -L(ライブラリの階層)
3
4  //Rust 言語組み込み済み C 言語プログラムの起動
5  sudo LD_LIBRARY_PATH=(ライブラリの階層) ./(
       C 言語プログラム)
```

4.2 Prusti による Rust 言語の検証

本節では，Rust 言語で構成したリストマップが 3.3 項で定義した条件を満たすことを検証する手法について説明する．

4.2.1 リストマップの構造と実行可能な操作

まず，検証の手法を説明するにあって本研究で実装したリストマップの構造を図 4 に示す．本リストマップは，u64 型のタプルを格納するために 2 つに連なった enum 型の Link を ListMap 構造体が管理することで実装される．次に，本リストマップで実装される操作について説明する．

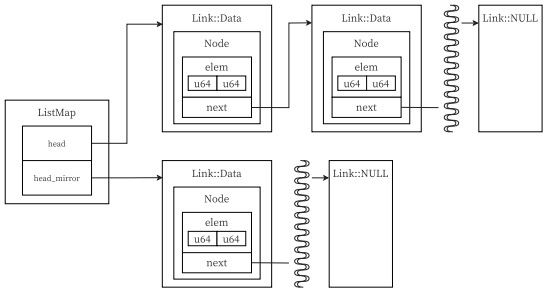

図 4 本リストマップの構造

生成: gen()

本リストマップは，ListMap 構造体の head と head_mirror それぞれがデータを格納していないことを示す Link::NULL を指すことによって生成される．

追加: insert(key, value)

key と value のペアを追加する為に該当データを格納した Link を作成した後，ListMap 構造体の head-/head_mirror が一番初めに指す Link との間に割り込むことで追加を行う．追加時に head が指す Link には key と value のペア，head_mirror が指す Link には head と逆に value と key のペアを格納することによって，図 3 のような NAT の双方向の変換を可能にする．

削除: remove(key, value)

ListMap 構造体の head と head_mirror が指す Link に対して，格納するデータが該当するものであるか確認し (head に対しては key と value のペア，head_mirror に対しては value と key のペア)，そうであればデータを格納する Link::Data ごと消去した後，前後の Link を繋ぐことで削除を行う．この操作は，連なる後続の Link 全てに対しても同様に行われる．

参照: get(key), get_mirror(value)

ListMap 構造体の head と head_mirror が示す Link に対して，格納するデータが該当するものであるか確認し (head に対しては key をキーにもつペア，head_mirror に対しては value をキーにもつペア)，そうであればその値を返し，そうでなければ後続の Link に対して同様の操作を行なっていく．この操作は，Last In First Out(LIFO) 方式のように直近に格納された値が参照されることとなる．また，TrustedOption という enum 型に Some(u64) と None を定義することで値を格納し，戻り値とする．

4.2.2 仕様の検証手法

次に，上記の操作を踏まえて Prusti による仕様の検証手法を示す．まず初めに Prusti で Rust 言語の検証を行うために，その設定ファイルである Cargo.toml と Rust 言語のプログラムに対してソースコード 2 のように記載を行う．

Prusti は Rust 言語に対して事前条件や事後条件，ループ不変条件などを定義することでプログラムがその条件を満たして動作するか検証することができる．また，異常終了である panic!や整数オーバーフローなどの検知も標準で搭載されているが，本論文では事前条件と事後条件を用いることでリストマップの仕様について検証を行う．事前/事後条件は関数が呼び出される前/後に満たされるべき条件であり，それぞれ#[requires(事前条件)] と#[ensures(事後条件)] を用いることで Rust 言語関数の上段に定義することができる．上記条件には，演算子や限定子，関数の戻り値などを用いることができるが，利用できる関数は純粋関数に限られる．これらの定義を用いて 3.3 項の条件を検証すると以下のようになる．

ソースコード 2 Prusti の実行準備

```
1  //Cargo.toml に追記
2  [dependencies]
3  prusti-contracts = { git = "https://
       github.com/viperproject/prusti-dev.
       git"}
4
5  //Rust 言語プログラムの上部に追記
6  use prusti_contracts::*;
```

空のマップには何も格納されていない

本リストマップが空となる場合は，マップが生成される時と格納されたデータが全て削除される時である．ただし，格納されたデータが全て削除されるという事象は追加と削除が行われることによって生じるため，追加と削除が正しく行われることが保証できれば検証する必要はないと考えられる．従って，検証の必要があるのはマップの状態推移の原点となる生成時であり，4.2.1 項のリストマップの構造に則ると満たすべき条件は次で定義される．

$$\forall k, s \;\; s.gen() \rightarrow s.get(k) == TrustedOption :: None \;\; \&\& \\ s.get\_mirror(k) == TrustedOption :: None$$

上記の条件を実際に Rust 言語上に定義したものをソースコード 3 に示す．

ソースコード 3 new 関数の検証

```
1  #[ensures(forall(|i: u64| (result.get(i).
       eql(TrustedOption::None))))]
2  #[ensures(forall(|i: u64| (result.
       get_mirror(i).eql(TrustedOption::None
       ))))]
3  pub fn new() -> Self {
4    ListMap {
5      head: Link::NULL,
6      head_mirror: Link::NULL
7    }
8  }
```

new 関数は，ListMap 構造体に対して定義され C++言語のクラスのような動作を行うことができ，gen 関数によって本関数が呼ばれることでリストマップが生成される．result は本関数の戻り値を表しており，マップ生成後に中身が存在しないことを確認するには get 関数で取得した値に対して，等号を用いることが簡潔であるが，標準で実装されている “==” 演算子は純粋関数である保証ができないため，独自の eql 関数を用いる．eql はソースコード 4 に示すように enum 型の TrustedOption に対して等号を定義する関数であり，#[pure] によって純粋関数であることを示すことで利用できる．

マップの追加，削除についても同様に検証を行うため，以降ではコードを省略して掲載する．

ネットワークの対応情報が追加された際に，実際に双方向で追加されている

本リストマップへのデータの追加は ListMap 構造体に定義される insert 関数によって実行される．insert 関数は，key と value のペアを引数にとり head と head_mirror へ，それぞれ逆方向にデータを格納することで NAT における双方向の変換を可能にする．従って，この関数で満たされるべき条件は次のように定義される．

ソースコード 4 eql 関数の定義

```
1  #[pure]
2  pub fn eql(&self, value: TrustedOption) ->
       bool {
3    match self {
4      TrustedOption::Some(i) => {
5        match value {
6          TrustedOption::Some(j) => *i == j,
7          TrustedOption::None => false,
8        }
9      }
10     TrustedOption::None => {
11       match value {
12         TrustedOption::Some(_) => false,
13         TrustedOption::None => true,
14       }
15     }
16   }
17 }
```

$$\forall k, s, v \;\; s.insert(k, v) \rightarrow \\ s.get(k) == TrustedOption :: Some(v) \;\; \&\& \\ s.get\_mirror(v) == TrustedOption :: Some(k)$$

上記の条件を実際に Rust 言語上に定義したものをソースコード 5 に示す．

ソースコード 5 insert 関数の検証

```
1  impl ListMap {
2    #[ensures(self.get(elem_k).eql(
         TrustedOption::Some(elem_v)))]
3    #[ensures(self.get_m(elem_v).eql(
         TrustedOption::Some(elem_k)))]
4    pub fn insert(&mut self, elem_k: u64,
         elem_v : u64) {...}
5  }
```

ネットワークの対応情報が削除された際に，実際に双方向で削除されている

本リストマップへのデータの削除は ListMap 構造体で定義される remove 関数によって実行される．remove 関数は，key と value を引数にとり head と head_mirror に対してそれぞれ key と value をキーにもつデータの削除を行う．従って，この関数で満たされるべき条件は次のように定義される．

$$\forall k, s, v \;\; s.remove(k, v) \rightarrow \\ s.get(k) == TrustedOption :: None \;\; \&\& \\ s.get\_mirror(v) == TrustedOption :: None$$

上記の条件を実際に Rust 言語上に定義したものをソースコード 6 に示す．

Prusti は，マップの状態が変化した後も TrustedOp-

ソースコード 6　remove 関数の検証

```
impl ListMap {
    #[ensures(self.check(elem_k) == false
        )]
    #[ensures(self.check_m(elem_v) ==
        false)]
    pub fn remove(&mut self, elem_k: u64,
        elem_v: u64) {...}
}

impl Link {
    #[ensures(self.check(elem) == false)]
    fn remove(&mut self, elem: u64) {...}
}
```

tion::Some(_) を用いた事後条件の検証が可能であったが，TrustedOption::None に関してはマップの状態が変化していない gen 関数での検証のみにとどまった．従って，上述の get 関数ではなく新たに定義した check 関数を用いることで検証を行う．check 関数はソースコード 7 に示されるように get 関数とは戻り値が異なり，値があった場合には true，なかった場合には false といった bool 型を返す．

ソースコード 7　check 関数の定義

```
impl ListMap {
  #[pure]
  pub fn check(&self, elem: u64) -> bool
      {...}
  #[pure]
  pub fn check_mirror(&self, elem: u64) ->
       bool {...}
}

impl Link {
  #[pure]
  fn check(&self, elem: u64) -> bool {...}
}
```

4.2.3　Prusti の利用制約

以上のように，事前条件と事後条件を駆使しながら関数の振る舞いを定義することによってリストマップが正しく実装される検証を行なったが，直感的な等号による定義や一見自明な条件に対しての検証が行えない場合が存在した．従って，本論文で障害となった仕様を現時点での Prusti の利用制約として以下に列挙する．

- Rust 言語の標準ライブラリで定義されている関数に対して，純粋関数であるか自身で判断し，定義し直す必要がある
- 標準ライブラリで定義される Option 型に対して等号を用いた条件の定義を行なった場合，検証が行えない場合があり，自身で Option と同様の働きをする TrustedOption のような構造体を定義する必要がある
- TrustedOption 構造体においても Some では検証が可能であるにもかかわらず，None では検証が行えない場合が存在する
- 複雑な関数に対して検証を行なった場合，その関数を利用して別の関数の検証条件を定義するとエラーとなる
- リストマップにある値 i を格納したのちに get 関数を呼ぶと，TurstedOption::Some(i) が返ってくるがその戻り値が TrustedOption の Some であることを保証できない
- u64 整数などが格納されるタプルといったの 1 重のデータ構造は検証可能であっても，同データを多重の配列やタプルに変更すると検証が行えなくなる

上記のような制約のほとんどは Prusti の内部エラーによるものであるため，検証条件を Viper で扱える形式に変換する Prusti の仕様の改善が望まれる．

4.3　本 NAT の機能

本 NAT は，LinuxOS が搭載される汎用 PC において NIC のドライバを DPDK 専用のものに交換した後に，当該プログラムがユーザー空間で実行されることでその機能を発揮する．初期設定では，2 つの NIC 間でパケットを転送するように動作するため，逐次 TCP 通信によってマップ管理 App からパケットの変換情報を更新する必要があり，当該 App は本 NAT プログラムが終了するまで入力を受け付け情報を反映させることができる．

また，本 NAT は検証の制約によりリストマップにタプルや配列を格納できないため，IP アドレスとポート番号を u64 の 64 ビット符号なし整数に変換することで情報の管理を行う．IP アドレスを表現するには 8bit×4 の計 32bit が必要であり，ポート番号には 16bit が必要であるため 48bit 以上で最も小さい整数である 64bit 整数を用い，図 5 に示す手法で情報を変換したのちマップへ格納する．ただし，ポート番号を持たない IPv4 パケットに対してはポート番号 0 を割り当てることとする．

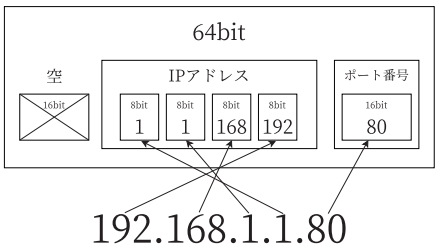

図 5　IP アドレスとポート番号の 64bit 整数への変換

5.　評価

5.1　定性的評価

本研究において実装した NAT の強みとしては，独自の

ツールや専門的な証明技術，記述言語を用いることなく Rust 言語による構築と Prusti による検証によりメモリ安全性と仕様に従った動作が保証される点である．

ここで，既存の NF 検証システムと本研究の検証手法について次の項目をもとに比較を行う．まず，表 1 に示されるように既存研究と本研究ともに，DPDK や AF_XDP といった高速パケット処理フレームワークが実装可能な C 言語でのパケット通信が可能である点が挙げられる．一方で，既存研究は独自の言語やツール，複雑な証明手法などを用いることによって検証を行うのに対して，本研究では既存の Rust 言語とその検証器である Prusti を用いることで専門的な技術の会得なしに検証が可能である．また，Vigor と本研究は静的な検証のみを対象としているのに対して，Aragog は実行時に検証を行えるため，複雑でスケールするネットワークにおいても検証を行うことができる．最後に，本研究では Rust 言語を用いることでメモリ管理の仕組みによってメモリ安全性を満たすことが保証できるため，Vigor のシンボリック実行などに比べて，検証のコストを抑えつつ，より高いレベルで安全性を実現できる．

表 1 定性的な比較評価

	VigNAT[11]	Aragog により検証された NAT[4]	本 NAT
C 言語上での実装	○	○	○
システム利用の簡易性	×	×	○
実行時検証	×	○	×
メモリ安全性	△	×	○

5.2 機能評価

本節では，本 NAT の機能評価として実際にソフトウェア上でパケットの変換を行うことで異なるネットワークでの通信を可能にした実行結果を示す．まず，図 6 に示すような本 NAT を実装したネットワーク環境について説明する．

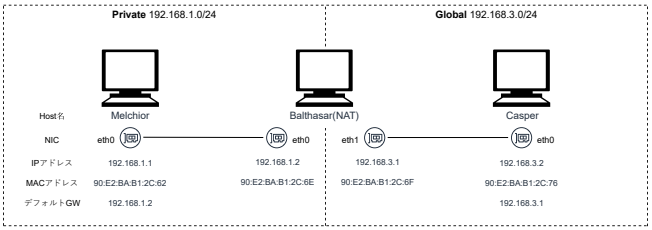

図 6 本 NAT の実装ネットワーク環境

Melchior の eth0 と Balthasar の eth0 は同じ 192.168.1.0/24 のプライベートネットワーク上に存在し，Balthasar の eth0 と Casper の eth0 は同じ 192.168.3.0/24 のグローバルネットワークに見立てた別ネットワークに存在する．Melchior と Casper は別ネットワークに存在するため Balthasar を介してパケットの伝達を行う必要があり，パケットの転送だけでは通信を行うことができない．したがって，本 NAT を用いることでパケットの IP アドレスとポート番号を変換し，ネットワークを越えた端末同士の通信を可能にする．

上記の環境を踏まえて，本 NAT が動作するための準備を行なったのち，実際に動作する様子を示す．本 NAT は ARP によるアドレス解決機能を搭載しないため，手動で IP アドレスと MAC アドレスの紐付けを登録した後に，パケットの変換時に宛先 MAC アドレスを適切な宛先に変更する機能を別途導入する必要がある．以上の操作を終えることで NAT は機能し，その実行結果をソースコード 8，9 に示す．

ソースコード 8 Melchior から Casper への Ping コマンド (ICMP パケット)

```
1  host@Melchior:~$ ping -I eth0 192.168.3.2 -c
2  PING 192.168.3.2 (192.168.3.2) 送信元 192.168.1.1 eth0: 56(84) バイトのデータ
3  64 バイト応答 送信元 192.168.3.2: icmp_seq=3 ttl=64 時間=0.703ミリ秒
4  64 バイト応答 送信元 192.168.3.2: icmp_seq=4 ttl=64 時間=0.785ミリ秒
5  (以下略)
6  --- 192.168.3.2 ping 統計 ---
7  送信パケット数 10，受信パケット数 8，パケット損失 20%，時間 9222ミリ秒
8  rtt 最小/平均/最大/mdev = 0.601/0.717/0.796/0.065ミリ秒
```

ソースコード 9 Casper における tcpdump コマンドによるパケットのキャプチャ (ICMP パケット)

```
1  host@Casper:~$ sudo tcpdump -i eth0 -nn
2  tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
   listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
3  IP 192.168.1.1 > 192.168.3.2: ICMP echo request, id 177, seq 1, length 64
4  IP 192.168.1.1 > 192.168.3.2: ICMP echo request, id 177, seq 2, length 64
5  IP 192.168.3.1 > 192.168.3.2: ICMP echo request, id 177, seq 3, length 64
6  IP 192.168.3.2 > 192.168.3.1: ICMP echo reply, id 177, seq 3, length 64
7  (以下略)
```

ソースコード 8 は，本 NAT の動作する環境で Melchior から Casper に ping を打った結果であり，送信された 10 パケット中，8 パケットに対して ICMP 応答が返ってきていることがわかる．このとき，2 パケットが送信された後，マップの追加により 192.168.1.1 に対する IP アドレスの変換が有効になっており，実際にソースコード 9 から送信元 IP アドレスが 192.168.3.1 に変換されていることがわかる

(逆方向も同様).

5.3 パフォーマンス評価

表 2 ping コマンドによる応答速度計測

	最小 (ms)	平均 (ms)	最大 (ms)	標準偏差
RawSocket	0.694	1.142	1.480	0.086
DPDK	0.241	0.723	0.866	0.065

上記計測結果より，DPDK による実装はネットワークスタックを利用する RawSocket に比べ，平均で 0.419ms 早く ping 応答を返していることがわかる．この差は RawSocket での応答時間の約 24 %にあたり，DPDK でのパケット通信は，従来のカーネルスタックを用いた通信に比べ，応答速度を 3/4 に短縮することを可能にした．また，DPDK は RawSocket に比べて応答速度の標準偏差が小さく，通信が安定していることがわかる．

6. まとめ

本論文では，我々の提案した Rust 言語，DPDK，Prusti を用いて設計される検証可能な NF について，NAT を例として実際に実装を行なった．3 章では，本 NF の概要を述べた後，NAT を実装する上でマップが満たすべき条件について定義した．4 章では，設計を基に実際に DPDK フレームワークが実装される C 言語から Rust 言語を呼び出す手法，前章で定義した条件について Prusti を用いて実際に検証する手順を明確にし，現在の Prusti の制約を列挙することで今後改善が望まれる仕様を明らかにした．

本研究の成果として，Rust 言語を用いたメモリ安全性，Prusti を用いた仕様検証，DPDK による高速パケット処理を実現する NF を提案することで，専門的な知識を要することなく，簡易な表現を用いた NF の検証が可能となった．また，パケット変換とマップによるデータの管理が行える NAT を例として実装することで，同様のデータ構造を持つ他の NF への拡張を可能とした．

今後の課題として，DPDK といったパケット通信部分の Rust 言語による実装，他の NF ににおける本手法の実装，より複雑なデータ構造を検証するための Prusti の性能向上が挙げられる．

7. 謝辞

本研究の一部は文部科学省「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)」，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」，さらに JSPS 科研費 JP21H034438 の助成を受けています．

参考文献

[1] Intel. DPDK: Data Plane Development Kit, 2021. https://www.dpdk.org/.

[2] 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.

[3] 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, Shanghai, China, October 28-31, 2017*, pp. 599–613. ACM, 2017.

[4] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pp. 701–718. USENIX Association, 2020.

[5] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh R. Iyer, Matteo Rizzo, Luis Pedrosa, Katerina J. Argyraki, and George Candea. Verifying software network functions with no verification expertise. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pp. 275–290. ACM, 2019.

[6] viperproject. Prusti, 2021. https://github.com/viperproject/prusti-dev.

[7] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, Vol. 6617 of *Lecture Notes in Computer Science*, pp. 41–55. Springer, 2011.

[8] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pp. 209–224. USENIX Association, 2008.

[9] Kohei Hosoya, Yuuki Takano, and Atsuko Miyaji. Implementation and verification of software bridges with filtering mechanisms in rust. In *SCIS 2022: The 39th Symposium on Cryptography and Information Security*, 2022.

[10] eqrion. cbindgen, 2021. https://github.com/eqrion/cbindgen.

[11] 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.