Computer Security Symposium 2023
30 October - 2 November 2023

Rustで記述されたLinuxカーネルドライバの安全性検査に向けて

青島 達大[1,a)]　大月 勇人[1]

概要：Rustは，メモリ安全で効率の良いプログラミングが実現できることを標榜するプログラミング言語である．OSやWebブラウザ等の基盤システムにおいて，その採用が広がっている．特に，Linux 6.1以降，Rustで書かれたカーネルモジュールが利用できるようになった．本論文では，将来的にRustでカーネルモジュールが記述されるようになった時代でも潜む脅威について議論する．まず，Rustにおける所有権と借用の概念を確認し，Rustが保証する安全性とその仮定を整理する．これらの仮定は，Rustにおけるunsafeコードが満たすべき条件を与える．具体例として，Rust-for-Linuxプロジェクトで試験実装されているNVMeドライバを題材に，検査を行った結果を報告する．また，unsafeコードを使わなくとも，提供されたRust側のインタフェースのみの利用でも生じるメモリ脆弱性を残してしまう危険性があることも示す．

キーワード：Rust，Linuxカーネル，安全性検査，NVMe

Towards the Safety Checking of Linux Kernel Drivers Written in Rust

Tatsuhiro Aoshima[1,a)]　Yuto Otsuki[1]

Abstract: Rust is one of programming languages advocating to realize memory-safety and memory-efficient programming. It has been adopted in several important foundation system such as OS and Web browsers. Linux kernel since version 6.1 supports to load kernel modules written in Rust. In this paper, in foreseeable future such that many Linux kernel modules are written in Rust, we reveal what kind of vulnerabilities may be contained in such Rust kernel modules. We review the ownership and borrowing of Rust, then clarify the safety ensured by Rust and its assumptions. These assumptions gives some of conditions for Rust unsafe code to satisfy. As an example, we will check the experimental NVMe driver proposed by the Rust for Linux project, and report some results. We found a risk that some kind of memory vulnerabilities will be left as only using safely provided Rust interface without any unsafe code.

Keywords: Rust, Linux kernel, safety checking, NVMe

1. はじめに

メモリ脆弱性を含まない効率の良いプログラムを実装する上で，Rust [1] というプログラミング言語が注目されている．Rustは，メモリ管理のために，ガベージコレクションを含む追加的なランタイム実装を必要とせず，型システムにより，プログラムにメモリ勢弱性が存在しないことを，事前に検査できると期待されている．Rustは2012年に最初のプレリリースバージョンが公開された比較的新しい言語であるが，Webブラウザ Mozilla Firefoxによる採用 [2] に始まり，Android [3]，Linuxカーネル [4]，Microsoft Windowsカーネル [5] 等の主要OSへの採用も進んでいる．

しかしながら，Rustの型システムによる，メモリ脆弱性が存在しないことに対する保証は，暗黙の仮定の上に成立している．例えば，標準ライブラリに含まれるunsafeコー

[1] NTTセキュリティ・ジャパン株式会社
NTT Security (Japan) KK
[a)] tatsuhiro.aoshima@global.ntt

© 2023 Information Processing Society of Japan

ドや，他の言語で書かれた既存のコードとやり取りするための Foreign Function Interface (FFI) の実装に含まれる unsafe コードにメモリ脆弱性が存在してはいけない．Rust における unsafe コードは，明示的に “unsafe” ブロックで隔離されるが，その中における安全性を型システムは保証しない．例えば，Rust の標準ライブラリ [6] は，可変長配列 “Vec<T>” を unsafe コードにより注意深く実装することで，バッファの確保や拡張，範囲外アクセスの検査を安全に実現している．OS や Web ブラウザ等は，独自に最適化されたオブジェクトのメモリ構造や，システム外部とのやり取りを行う必要があるため，Rust を使えば必ずメモリ安全なプログラムが作れるとは限らない．

Rust の unsafe コードの存在は，主要な OS や Web ブラウザにおける採用に向けて，様々な議論を呼び起こす元となってきた．例えば，Linux カーネルでは，開発ルール [9] が提案されており，Rust で記述できる範囲はドライバと末端モジュールに限定している．これは，既存の C 言語で記述されたカーネルコードの上に位置するが，Rust のコードから直接アクセスできないようにすることで，Rust の型システムを活用して，安全なインタフェースを提供するためである．これにより，メモリ脆弱性やデータレース，ロジックバグがなくなることを期待している．ただし，そのインタフェース自体を実装するために必要な unsafe コードの安全性は，Rust の型システムにより保証されないので，安全性に関する事前条件や型不変性をコメントとして明文化する運用を行うことも規定している．

以上が基本的なルールだが，安全性検査は開発者やメンテナにより，手動で実施されている．開発者が具体的にどのような観点でどこまで記述すれば良いかは明確でなく，また，メンテナもどのような基準で確認すべきかは明文化されていない．よって，インタフェースの実装自体に脆弱性が潜む可能性があり，将来的に開発者やメンテナにも想定できなかった方法で利用されると，ドライバに脆弱性が残ってしまう可能性もある．

そこで本論文では，将来的に OS や Web ブラウザ等の基盤システムが，Rust で実装されるようになった時代において，どのようなメモリ脆弱性が潜むのか，その脅威について明らかにする．具体例として，Linux カーネルドライバを Rust で記述しようと試みる実験プロジェクト Rust for Linux [7] で試験実装が進められている NVMe デバイスドライバ [8] を検査する．

本論文の構成は，次の通りである．2 章で，Rust の標準ライブラリを例に，メモリ安全性を保証する上で欠かせない概念となる，所有権と借用について整理する．3 章で，所有権と借用の概念に基づいて，Rust が保証する安全性から，防ぐことのできるメモリ脆弱性を示し，その実現における仮定を整理する．これにより，unsafe コードをどのように検査すれば良いか，という観点が明らかになる．4 章で，実際に Rust for Linux による NVMe ドライバの検査を行い，その課題や問題点を示す．ここで，Rust 側に提供された安全なインタフェースのみを用いて，解放後使用の脆弱性が残ってしまう可能性があることも指摘する．以上の議論をまとめ，5 章で，今後 Rust により基盤システムを効率良く，そして安全に実装するためにはどのような取り組みが必要か，提言を行う．また，本論文の公開が，倫理的配慮の点においても問題がないことも説明する．最後に，6 章でまとめを行う．

2. Rust の基礎

本章では，Rust の標準ライブラリ [6] を具体例として，メモリ安全性を保証するために最も欠かせない概念である，所有権と借用について説明する．

2.1 所有権と借用

型 T のデータ本体 (メモリ表現そのもの) を，**オブジェクト**と呼ぶ．プログラムにおける制御パスの集合を，オブジェクトの**生存期間** (lifetime) と呼び，小文字のアルファベットを用いて，'a, 'b 等と書く．生存期間 'a, 'b に関して，'a が 'b を含むとき，'a: 'b ('a outlives 'b) と書く．変数 v が，オブジェクト o: T をある生存期間 'a で所有できる権利を**所有権** (ownership) といい，v: T<'a>と書く．オブジェクトを所有している限り，オブジェクトの読み書きと所有権の移動ができるものとする．所有権を別の変数 u へ渡した後は，変数 v が持つ所有権は消失する (ムーブ意味論)．生存期間の終了時点において，対応する所有権が放棄され，同時にオブジェクト (データ本体) は解放される．具体的には，drop メソッドを呼び出し，型 T に対応する片付けの処理を実装できる．

所有権を有する変数 v から，読み書きできる権利を借りた状態を**可変参照** (mutable reference) rm: &mut 'b T<'a> とする．また，読み込みだけができる権利を借りた状態を**不変参照** (immutable reference) ri: &'b T<'a>とする．このとき，不変参照と可変参照は，次の関係を満たさなくてはいけない．

- **(参照の生存期間)** 不変参照 ri や可変参照 rm が有効なのは，所有権の生存期間の内側に限られる．つまり，'a: 'b でなくてはいけない．
- **(生存期間の入れ子関係)** 不変参照 ri または可変参照 rm から新しい不変参照 ri2: &'c T<'a>を作るとき，ri2 の生存期間は ri や rm の生存期間の内側に限られる．つまり，'b: 'c でなくてはいけない．
- **(可変参照の一意性)** 不変参照は同時に複数存在できるが，可変参照が存在するとき，それは一意であり，また他に不変参照が存在してはいけない．

© 2023 Information Processing Society of Japan

```
1  fn main() {
2    let mut v = Vec::new();
3    v.push(1);
4    println!("length: {}", v.len());
5    process(v);
6    //ERROR: v.push(4);
7  }
8  fn process(v: Vec<usize>) {
9    v.push(2);
10   for x in v.iter() {
11     println!("element: {}", x);
12     //ERROR: v.push(3);
13 } }
```

図 1 可変長配列型を使用する Rust コードの例
Fig. 1 An example code in Rust using Vec.

2.2 可変長配列型

所有権と借用の規則に従うことで，どのようにメモリ管理を安全に実装できるかを説明するために，Rust の標準ライブラリに含まれる可変長配列型 Vec<T>を考える．**図 1** に，説明のためのサンプルコードを載せる．

2 行目で Vec<i32>型のオブジェクトを確保し，長さ 0 で初期化する．ここから，変数 v はこの配列への所有権を獲得する．3 行目は，`Vec::<i32>::push(&mut v, 1);` の糖衣構文である．よって，変数 v から可変参照を取得して，このオブジェクトへ要素 1 を追加する．4 行目は，`Vec::<i32>::len(&v)` の糖衣構文であり，長さを取得するメソッドを呼び出す．このとき，変数 v から不変参照を取得すべく，3 行目で取得した可変参照の生存期間は 4 行目の直前で終わるものと型システムは推定する．

5 行目で process 関数へオブジェクトの所有権を変数 v から第一引数へ渡す (値渡しによるムーブ意味論)．よって，6 行目で，変数 v から可変参照を取る操作は，所有権を持たなくなった変数 v からの借用となるため，型検査により不正とみなされる．これにより，移動後の変数にアクセスしてしまうメモリバグを防ぐことができる．

process 関数は，第一引数経由でオブジェクトの所有権を受け取る．10 行目で不変参照を取得し，配列の各要素への不変参照を順に処理するイテレータを作る．11 行目で，各要素への不変参照を利用して，表示 (読み込み) を行う．ただし，12 行目で，第一引数から可変参照を取り出す操作は，可変参照の一意性に反するため，型検査により不正とみなさせる．この場合，逆に認めてしまうと，配列中のバッファの大きさが足りないときに，別の場所へ拡張される形で再確保されるため，イテレータが保持している古いバッファへの参照が残ってしまう．よって，可変参照の一意性により，イテレータを利用した解放後使用のメモリバグを防ぐことができる．

2.3 参照カウンタ

可変参照の一意性は，安全性に対する十分条件となるが，一般には強すぎる．例えば，グラフ構造のように，あるオブジェクトが複数箇所から参照されて，書き換えられるようなコードを書くことができない．そこで，「参照カウンタへの参照を所有する」という発想を利用する．

Rust の標準ライブラリでは，Rc<T>型として実装されている．作成したオブジェクトを Rc<T>型で包むとき，また，参照を増やすために clone メソッドを呼ぶときに，必ず参照カウントが増加するように実装されている．また，Rc<T>型のオブジェクトへの所有権が放棄されたとき，必ず参照カウントが減少するように，drop メソッドが実装されている．型内部のフィールドは公開されない限り，外部から直接アクセスできないため，参照カウントの整合性は維持される．しかしながら，Rc<T>型の実装自体は，参照カウントが 0 になったときに限り，手動で解放する必要があるため，unsafe コードとして注意深く実装されている．

2.4 NULL 安全性とエラー処理

Rust の Option<T>型により，NULL 安全性を保証できる．例えば，C 言語において，メモリ確保を行う malloc 関数は，十分な領域の確保に成功したら，そこへの正当なポインタを返し，失敗したら，NULL ポインタを返す．共に同じポインタ型であるため，NULL チェックを行ったかどうかを型システムで検査するのは，一般には難しい．Rust では Option<T>型を用いて，成功したら，Some(v) を返し，失敗したら，None を返すように実装する．戻り値を型 T の値 v として利用すると，型が合わないので，NULL チェックが抜けていることを確認できる．

エラー処理において，その内容も保持したい場合，Result<T, E>型を使って，成功したら，Ok(v) を返し，失敗したら，Err(e) を返すように実装する．戻り値を型 T の値 v として利用すると，型が合わないので，エラーチェックが抜けていることを確認できる．

3. Rust の unsafe コードに対する検査

本章ではまず，Linux カーネル開発において，Rust の unsafe コードを検査するためのルールを整理する．現状の unsafe コードに対するコメントに依存した検査は不十分であるため，2 章で説明した所有権と借用の概念に基づき，Rust が保証する安全性と unsafe コードが満たすべき十分条件を示す．最後に，その十分条件を満たすモデルを用いて検査を行う考え方を説明する．

3.1 Linux カーネル開発におけるルール

[9] によれば，Linux カーネルの開発において，Rust で記述されたコードが直接，C 言語で記述されたコードを呼び出すことがないように，安全なインタフェースを設計す

© 2023 Information Processing Society of Japan

るルールを設けている．このとき，そのインタフェース自体の実装や，適切なインタフェースが存在しない機能を直接利用するために，unsafe ブロックが利用されることがある．しかしながら，unsafe ブロックの安全性は Rust の型システムによって検査されない．そこで，その安全性条件をコメントとして記述するルールも設けられている．

理想的には，自然言語によるコメントとその人手による確認に頼らず，プログラム解析の技術を併用した自動化が望ましい．例えば，Rust の Miri [10] を利用して unsafe ブロックを検査できるが，C 言語で記述された既存のコードによる影響を考慮できないことが指摘されている [11]．メモリバグを検知するために，Kernel Address Sanitizer (KASAN) [12] が存在するが，執筆時点 (2023 年 7 月) の最新安定版 (6.4) や Rust for Linux の開発ブランチでも対応予定となっており [13]，Rust の unsafe コードに対して適用できない．

3.2 Rust が保証する安全性

3.1 節で整理したように，Linux カーネルの開発において，Rust の unsafe コードに潜むメモリバグを検知するには，人手による検査が最善となっている．そこで，本論文では，2 章で整理した所有権と借用の概念に基づいたモデルを構築し，そのモデルに従った安全性検査を実践する．Rust は所有権と借用の概念を埋め込んだ型システムと標準ライブラリを含むランタイムを合わせることにより，次のような機能を提供する．

- **(生成と初期化)** 型システムにより，生成直後から初期化前までの不完全な状態が存在しないことを条件とする Resource Acquisition Is Initialization (RAII) に従うことを検査する
- **(時間的保証)** 型システムにより，所有権と借用の規則に従うことを検査する
- **(範囲の保証)** ランタイム実装により，添字の範囲を検査する
- **(ロジックの保証)** 型システムにより，(一部の既定型間を除いて) キャストが行われないことを検査する

(生成と初期化) により，アクセスされる前に，必ず正しい状態へ初期化されることが保証されるため，不正ポインタ使用や未初期化使用の脆弱性を防ぐことができる．次に，(時間的保証) により，参照前に，生存していることも保証されるため，二重解放も含む解放後使用やデータレースの脆弱性を防ぐことができる．さらに，(範囲の保証) により，参照される範囲が確保された範囲を超えないことも保証されるため，範囲外アクセスの脆弱性を防ぐことができる．最後に，(ロジックの保証) により，参照先のデータが意図した型に従うことも保証されるため，型混同の脆弱性を防ぐことができる．

3.3 unsafe コードに対する安全性仮定とその十分条件

しかしながら，3.2 節で示した安全性の保証は，暗黙の仮定の上に成立している．そこで，4 つの機能それぞれについて，その十分条件を示す．

- (生成と初期化) unsafe コード内部で生成された値が不正なものでなく，unsafe コード外部へ渡す前に初期化されること
- (時間的保証) unsafe コード内部で生成された値の参照関係を意識し，unsafe コード外部へ渡した参照が有効な限り，値が解放されないこと
- (範囲の保証) 算術オーバーフローやアンダーフロー，ゼロ除算に注意しつつ，サイズの計算を行い，ポインタや添字の範囲を検査すること
- (ロジックの保証) unsafe コード内部で生成された値の型を正しく特定し，一貫して利用すること

unsafe コード内部で生成された値は，生のバイナリ表現から変換されたものに限らず，Foreign Function Interface (FFI) 経由でやり取りされるものも含まれる．よって，FFI で受け取った値への参照を，解放後にアクセスしないことや，FFI で渡した値が，Rust における所有権と参照の意味論と一貫しており，期待される期間以上は必ず存在すること等も検査する必要があり，非常に困難なタスクとなる．なお，これらは十分条件であるため，これらの条件を満たさないが，結果として安全であるコードも考えられる．理解しやすく，検査を簡単にするために示した一例である．

以上より Rust は，unsafe コードがある種の仮定を満たせば，メモリ安全性を保証できるプログラミング言語となる．OS や Web ブラウザ等は，独自のオブジェクト表現や I/O 処理機構を持っていることも多く，それらを処理するための unsafe コードが出現することも多いと考えられる．本節で示した十分条件を確かめることで，unsafe コードを安全に実装できたことを確認できる規準となる．

3.4 オブジェクトの所有権モデルに基づく検査

2 章の具体例で示した通り，Rust の型システムにより，unsafe ブロックが存在しない限り，関数単位で安全性を検査できる．しかしながら，unsafe ブロックの中身は検査されないため，今のコードに存在する制御フローに限らず，任意のあらゆる unsafe でないインタフェースの利用と組み合わせても，安全性仮定に対する十分条件を満たすことを検証する必要がある．

そこで本論文では，所有権と借用の概念に基づき，オブ

© 2023 Information Processing Society of Japan

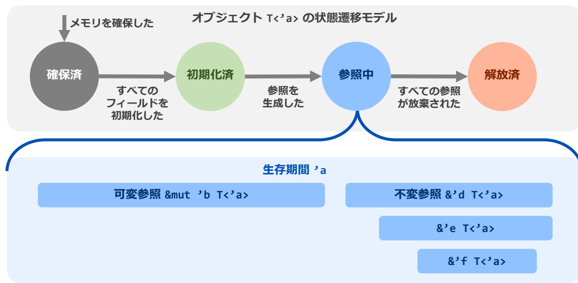


図 2　オブジェクトの所有権モデルを示す図
Fig. 2　An object ownership model.

ジェクトの状態を意識した簡単な抽象状態遷移モデルを設計し，それに基づいて検査を行う．オブジェクトは，2 章で述べた通り，メモリ上に確保されるデータ構造そのものとする．任意のオブジェクトは，(生成と初期化) と (時間的保証) に関する条件より，**図 2** の上側で示す通りの状態遷移モデルに従う必要がある．最初にオブジェクトが確保され，すべてのフィールドを初期化した後に，参照を生成できる．そして，すべての参照が放棄された後に解放することができる．

可変参照と不変参照は，図 2 の下側で示す通り，生存期間を軸に取って解釈できる．可変参照が存在する限り，他の参照が存在してはいけない．また，不変参照を複数作ったり，不変参照から他の不変参照を生成できるが，生存期間は生成元の生存期間に含まれていなくてはいけない．

この参照に関する関係は，オブジェクト中のフィールドへの参照を取る操作にも適用される．よって，複数のオブジェクトが参照関係にある場合，それらの生存期間はそれぞれ正しい方向で包含されていなくてはいけない．以下，オブジェクトの状態遷移モデルと生成された参照たちが満たすべき生存期間に関する制約を合わせて，**オブジェクトの所有権モデル**と呼ぶ．

4.　NVMe ドライバの検査例

3.4 節で示した検査手順に従って，Rust for Linux プロジェクト [7] で試験実装されている NVMe ドライバ [8] の検査を行い，問題が見つかった場合，より安全な形へ修正できることも示す．具体的にはまず，NVMe ドライバが利用する Linux カーネルの API を整理することにより，unsafe コードが満たすべき要件を明らかにする．次に，その要件に従い，オブジェクトの所有権モデルを設計する．最後に，そのモデルに基づき，ソースコードを検査する．

4.1　検査対象

今回の検査対象は，Non-Volatile Memory express (NVMe) という不揮発性補助記憶装置を接続するインタフェース規格 [14] のドライバ実装である．PCI Express デバイスとして動き，Direct Memory Access (DMA) の操作として制御を行う，メモリベースのトランスポートモデルにのみ対応している．なお，本ドライバについては，“Linux カーネルには既に NVMe デバイスドライバの高性能な C 実装が存在するため，即座に代替するものではない” と明言されており [15]，Proof of Concept に過ぎない．

Linux カーネルの開発において，`unsafe` ブロックの安全性に関するコメントを残すことがルール [9] となっているが，この実装には，記述のない箇所がある．また，インタフェースの整備も検討中であり，`unsafe` ブロックが NVMe ドライバ本体の実装にも一部出現している．

本論文では，NVMe ドライバの DMA 処理におけるメモリ安全性に焦点を絞って議論する．これは，カーネルモジュールや PCI デバイスドライバ，ブロックデバイスドライバとしての実装は，ドライバのロードやデバイスの登録時にしか実行されないコードであること，また，デバイスの除去時のコードが未実装であることから，より様々なコンテキストで実行される DMA 処理の検査に注力したことによる．

以下，必要であれば，コミット [8] 時点で変更されたファイルの名前と行番号のみを以ってして，“(filename.rs:123)” のようにソースコード中の箇所を提示する．ファイル名だけで伝わりにくい場合は，適切なレベルの相対パスで表記する．Rust におけるモジュールやメソッドのパス表現においても，同様の表記を採る．

4.2　DMA プールの所有権モデル

NVMe ドライバは DMA を利用して，デバイスとデータ本体をやり取りするため，カーネルが提供する DMA プールの API [16] を利用する．これは，DMA 上でコマンドバッファを管理するために利用されている．

DMA プールの処理に関するメモリ安全性を検査するために，DMA プールの所有権モデルを構築する．DMA プールの API に関する仕様はドキュメント [16] に従うものと仮定し，DMA プールを構成する要素をオブジェクトとして抽象化する．本論文では，API を提供する既存の実装に備わるバグは検査しない．また，インタフェースとして隔離され，その利用条件も正しく記述されているものと仮定するため，実装の詳細は考慮しない．

API のドキュメント [16] によれば，**図 3** で示す通りの所有権モデルが得られる．ここで，DMA プールは，関数 `dma_pool_create` で作成されるオブジェクトのことを指し，`dma_pool_destroy` 関数により解放される．DMA プールから領域を確保する関数 `dma_pool_alloc` と，その解放を行う関数 `dma_pool_free` を適切に利用するためには，領域確保元となる DMA プールへのポインタと，確保された領域を示す仮想アドレス，DMA ハンドルをまとめて三つ組 (オブジェクト) として扱う必要がある．本論文では，これを **DMA トリプル**と呼ぶ．以下，DMA トリプル

© 2023 Information Processing Society of Japan

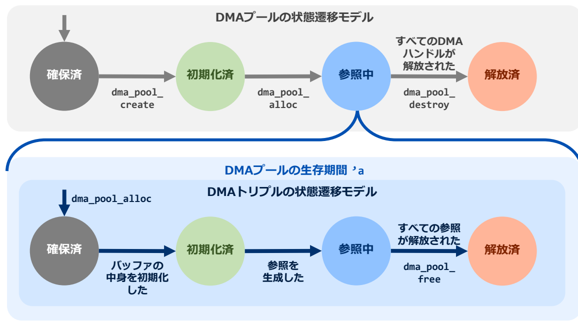


図 3 DMA プールと DMA トリプルの所有権モデル
Fig. 3 An object ownership model of a DMA pool and DMA triples.

の解放は，DMA プールへのポインタを利用して，仮想アドレスと DMA ハンドルを `dma_pool_free` へ渡し，プールへ返却する処理のことをいう．

本論文では，次の 3 点に絞って，NVMe ドライバの実装がこのモデルに従っていることを検査する．まず，DMA トリプル中のバッファの範囲を超えてアクセスしないことを検査する (4.3 節)．次に，DMA プールから確保された任意の DMA トリプルは必ずちょうど 1 回だけ解放され，その後に利用されないことを検査する (4.4 節)．最後に，DMA プールの生存期間は，そこから確保される任意の DMA トリプルの生存期間以上でなくてはいけないことを検査する (4.5 節)．

4.3 範囲外アクセスの検査

DMA プールから DMA トリプルを確保するために，Rust 側では `dma::Pool::try_alloc` 関数が提供されており，DMA トリプルは `CoherentAllocation<T, A>`型でラップされる．これは，DMA トリプルが指すバッファを単なるバイト配列でなく，型 `T` の配列型として解釈するように設計されている．よって，配列として，範囲外アクセスが行われないかを検査する必要がある．

まず，型 `T` の配列をひとつのバッファとして確保するように，チェック算術 `checked_mul` を用いて，オーバーフローしないようにサイズを正しく計算する (dma.rs:90)．また，DMA トリプルに加えて，配列の長さも記録する．例えば，`read` メソッドは，添字の検査を符号なし整数の比較として行い，範囲内にある場合は，丸め込みのポインタ算術 `wrapping_add` でアクセスして `Some(v)` を返し，そうでない場合は `None` を返す (dma.rs:164)．ここで，ポインタ算術が丸め込みで問題ないことは，サイズの計算が正しいことと，`dma_pool_alloc` が返した仮想アドレスにサイズを足したときにオーバーフローが起こらないという API が保証すべき仮定から分かる．他の読み書きを行うメソッドについても，同様に検査できる．

4.4 DMA トリプルの解放パス

`try_alloc` 関数は NVMe ドライバの実装において，`setup_prps` 関数における 2 箇所からしか呼ばれない (nvme_prp.rs:{72, 96})．DMA トリプルは `CoherentAllocation<T, A>`型としてラップされ，`drop` メソッドで適切に `dma_pool_free` が呼ばれる設計であるにも関わらず，`ManuallyDrop` 型と `ScopeGuard` 型で包み込み，DMA トリプルの中身を別々に取り出して，さまざまな箇所へ格納している．また処理に成功したら，`drop` メソッドが呼ばれないように `ScopeGuard::dismiss` メソッドを呼び出して (nvme_prp.rs:135)，手動で解放を行うようにする．よって，`setup_prps` 関数は `unsafe` ブロックを含まないが，DMA トリプルの所有権モデルを無視している．

DMA とブロックデバイスドライバの意味論を考えれば，DMA 処理が完了したときに解放を行うものと考えられる．ドキュメントとして明文化されていないが，本論文では，ひとつのリクエストオブジェクト (`blk_mq_queue_data` 構造体) は `blk_mq_ops` の `queue_rq` メソッドへ渡され，処理が終了した後，ちょうど一回だけ `blk_mq_ops` の `complete` メソッドに渡されるものと仮定する．よって，`queue_rq` メソッドから `complete` メソッドへのちょうど 1 回の遷移を考えれば良い．

まずは，`setup_prps` で確保処理において，エラーが発生せずに，正常にリクエストが発行される場合を考える．DMA トリプル `prp_list` を構成する各要素の伝播を **図 4** に示す．図において，三角矢印が参照関係，四角矢印が値のコピーによる代入関係を示す．関数 (❺と❻) の引数へ渡される場合も，値のコピーとみなせるため，四角矢印で表現した．また，DMA トリプルを包む `ManuallyDrop` や `ScopeGuard` 自体は `CoherentAllocation` 構造体だけを持つものとみなせるから，見やすさのために省略した．なお，ブロック I/O サブシステムによる処理のせいか，我々が試した範囲では再現できなかった，より大きなデータを処理するために，PRP リストを分割して，`pages` 配列へ格納していく処理 (nvme_prp.rs:95) の検査は省略した．

大まかな処理の流れは，次の通りである．

❶ `try_alloc` 関数を呼び出し，DMA トリプルを含む `prp_list: CoherentAllocation<T, A>`を確保する．`alloc_data` が DMA プールへのポインタで，`cpu_addr` が確保した領域の仮想アドレス，`dma_handle` が対応する DMA ハンドルを持つ．

❷ ❶の呼び出し元である `setup_prps` 関数において，ブロック I/O サブシステムが用意したデータ転送用のバッファの配列 `md.sg` を確保した領域へコピーする．このとき，DMA トリプルは各要素へ分解され，仮想アドレスは `md.pages[0]` へ，DMA ハンドルは NVMe のリクエストコマンド `cmd.prp2` へ代入される．DMA

プールへのポインタは，デバイスオブジェクトから毎回正しく取り出されるため，その追跡は省略する．

❸ ❷の呼び出し元である queue_rq メソッドで，pdu を初期化する．

❹ リクエストが完了したら，ブロック I/O サブシステムにより，complete 関数が呼ばれる．

❺ データ転送用のバッファの配列 md.sg を解放するために，dma_unmap_sg 関数を呼び出す．その長さは pdu.sg_count から取り出しているため，❷より，正しく解放処理を呼び出すことがわかる．

❻ PRP リストを解放するために，free_prps 関数 (nvme_prp.rs:12) を呼び出す．md.pages から配列を取り出し，pdu.page_count から長さを，first_dma から DMA ハンドルを，dma_pool から確保元の DMA プールを取り出しているため，❷より，正しく解放処理を呼び出すことがわかる．

最後に，setup_prps で確保処理においてエラーが発生すれば，ScopeGuard::drop メソッドにより，free_prps 関数が呼び出され，それ以前に確保した DMA トリプルがすべて適切に解放される．以上より，これらは我々による人手での解析に過ぎないが，この範囲では所有権モデルに従うことが分かる．

4.5 DMA プールの生存期間

DMA プールから DMA トリプルを確保する関数は，pub fn try_alloc(&self, atomic: bool) -> Result<CoherentAllocation<T, Self>>と宣言されている (dma.rs:107)．これでは, 戻り値の生存期間が DMA プールの生存期間で制限されないので，DMA プールの生存期間を超えて，確保した DMA トリプルを保持し続けることができてしまう．実際に簡単な再現コードを用意したところ，Rust の型検査は通るが，実行すると DMA トリプルの解放時に，DMA プールのスピンロックを取る箇所で，KASAN が解放後使用を検知した．このように，安全なインタフェースだけを用いて，DMA プールの解放後使用を引き起こすことができる．今回の NVMe ドライバではそのような利用パターンは見つけられなかったが，今後，unsafe コードを使ってないから，型検査が通る限り，どのように利用しても安全である，とは言い切れなくなる．

解決策としては例えば，DMA プールの生存期間 'a を戻り値の型である CoherentAllocation に生存期間を付けて，pub fn try_alloc(&'a self, atomic: bool) -> Result<CoherentAllocation<'a, T, Self>>とすれば良い．これにより，DMA プールの生存期間を超えて，戻り値となる DMA トリプルを保持しようとする箇所を型検査により，保守的に検知できる．

5. 議論

4 章の検査結果を踏まえ，今後 Rust を用いて，安全に Linux カーネルドライバを開発するには，どうすべきか整理し，提言としたい．特に，所有権モデルに基づく検査手法は，Linux カーネルドライバに限った話ではないため，OS や Web ブラウザ等の基盤システム特有のオブジェクト表現や I/O 処理に対する検査でも活かせる点もあるだろう．

5.1 DMA ハンドルの抽象化

DMA トリプルは CoherentAllocation<T, A>型として抽象化されているが，NVMe ドライバではその中身を分解し，DMA ハンドルを単に符号無し整数型として扱う．これにより，DMA プールから確保された PRP リストがどのような流れで解放されるのかを追跡することが困難となった．例えば，DMA ハンドルを構造体に包むことにより，単なる整数型と区別できる．また，適切なアクセサにより範囲を検査する等，Rust の型システムを活用して，DMA ハンドルのロジックに基づいた検査が可能となるように，設計を改善できるだろう．

5.2 コールバックインタフェースの抽象化

ブロックデバイスの API はコールバック型であるため，コールバックの完了処理までに，DMA ハンドルを含む情報から，Rust の参照であることが忘れられてしまうことは避けられない．将来的には，async/await によりコールバック機構を抽象化し，非同期な処理を簡潔に書けるようにする方向性も検討されている [15]．しかしながら，現状の DMA プールのインタフェースは，DMA プールと DMA トリプルの借用に関する規則に従わないため，本論文で指摘した通り，async/await を利用しても，安全なコードを書けるとは限らない．よって，DMA トリプルの所有権モデルを意識した async/await の設計が必要となるだろう．

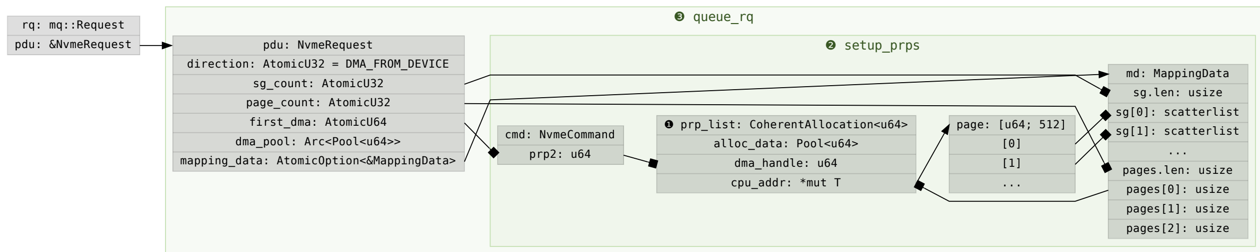
5.3 倫理的配慮

本研究で扱った NVMe ドライバはメインラインへマージされていない [7]．よって，Proof of Concept 段階の実装に対する脅威の検査に過ぎず，実社会における影響はないと考えている．むしろ本論文が，今後開発されていくであろう様々な Rust で記述されたコードの安全性検査において活用できる知見を提供するものと期待する．

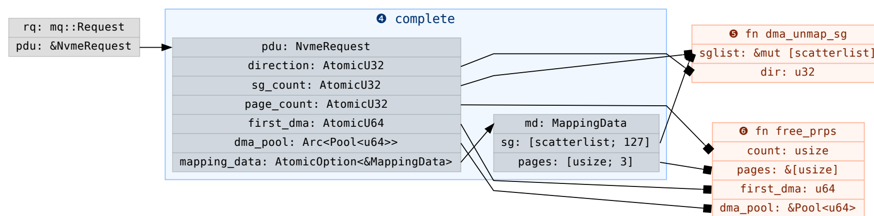
6. おわりに

本論文では，OS や Web ブラウザ等の基盤システムが Rust によって記述されても潜む脅威を分析した．所有権と借用の概念により，Rust が保証する安全性に対する仮定を明らかとなり，unsafe コードの検査条件を導出できた．

© 2023 Information Processing Society of Japan



(A) リクエストの生成 (queue_rq 関数)



(B) リクエストの完了 (complete 関数)

図 4 DMA トリプル prp_list を構成する各要素の伝播

Fig. 4 Propagation of each field in a DMA triple prp_list.

また，所有権モデルを設計し，Rust for Linux プロジェクト [7] で試験実装された NVMe ドライバ [8] を検査した．実際に，unsafe コードを利用することなく，安全な Rust 側のインタフェースのみを用いて，DMA プールの解放後使用を引き起こすことができることも確認し，その修正方法も示した．

Linux カーネル開発では，`unsafe` ブロックの安全性に関するコメントを残すルールがある [9]．C 言語で記述された既存のコードと相互運用されるため，sanitizer の利用が望ましいが，現時点では Rust と併用できない．今後は，`unsafe` ブロックの安全性を自動的に検査するための研究も求められるだろう．本論文の所有権モデルは，そのような研究で考慮すべき条件を与える．本論文が，Rust を用いた安全な基盤システム開発を実現するための研究や議論へ，具体的な示唆を与えることを期待する．

参考文献

[1] Rust Programming Language (online), available from 〈`https://www.rust-lang.org/`〉(accessed 2023-07-19).

[2] Mozilla,「Firefox 57」改め「Firefox Quantum」を発表～従来バージョンの 2 倍高速 - 窓の杜（オンライン），入手先 〈`https://forest.watch.impress.co.jp/docs/news/1083038.html`〉（参照 2023-07-19）.

[3] Rust の採用が進んだ「Android 13」，メモリ破壊バグは減少，脆弱性の深刻度も低下傾向 - 窓の杜（オンライン），入手先 〈`https://forest.watch.impress.co.jp/docs/news/1462573.html`〉（参照 2023-07-19）.

[4] LKML: Linus Torvalds: Linux 6.1-rc1（オンライン），入手先 〈`https://lkml.org/lkml/2022/10/16/359`〉（参照 2023-07-19）.

[5] Announcing Windows 11 Insider Preview Build 25905 — Windows Insider Blog (online), available from 〈`https://blogs.windows.com/windows-insider/2023/07/12/announcing-windows-11-insider-preview-build-25905/`〉 (accessed 2023-07-19).

[6] std - Rust (online), available from 〈`https://doc.rust-lang.org/stable/std/`〉 (accessed 2023-07-19).

[7] Rust for Linux (online), available from 〈`https://rust-for-linux.com/`〉 (accessed 2023-07-19).

[8] Update safety comments · metaspace/linux@7b7ca73 (online), available from 〈`https://github.com/metaspace/linux/commit/7b7ca733ab927acaa0e3c3600e11c3ba69e3bc51`〉 (accessed 2023-07-19).

[9] LKML: ojeda@kernel ...: [PATCH 00/13] [RFC] Rust support (online), available from 〈`https://lkml.org/lkml/2021/4/14/1023`〉 (accessed 2023-07-19).

[10] rust-lang/miri: An interpreter for Rust's mid-level intermediate representation (online), available from 〈`https://github.com/rust-lang/miri`〉(accessed 2023-07-19).

[11] Re: [TECH TOPIC] Rust for Linux - Marco Elver (online), available from 〈`https://lore.kernel.org/ksummit/YOgesjNqpsZNK5Gf@elver.google.com/`〉 (accessed 2023-07-19).

[12] The Kernel Address Sanitizer (KASAN) — The Linux Kernel documentation (online), available from 〈`https://docs.kernel.org/dev-tools/kasan.html`〉 (accessed 2023-07-19).

[13] 'rustc' wanted features & bugfixes · Issue #355 · Rust-for-Linux/linux (online), available from 〈`https://github.com/Rust-for-Linux/linux/issues/355`〉 (accessed 2023-07-19).

[14] Specifications - NVM Express (online), available from 〈`https://nvmexpress.org/specifications/`〉 (accessed 2023-07-19).

[15] Hindborg, A: Linux (PCI) NVMe driver in Rust, LPC 2022 (2022). (online), available from 〈`https://lpc.events/event/16/contributions/1180/attachments/1017/1961/deck.pdf`〉 (accessed 2023-07-19).

[16] Dynamic DMA mapping using the generic device — The Linux Kernel documentation (online), available from 〈`https://docs.kernel.org/core-api/dma-api.html`〉 (accessed 2023-07-19).

© 2023 Information Processing Society of Japan