A-003

Rust 言語における MPK を用いた外部関数呼び出しのプロセス内分離

In-process isolation of foreign function calls using MPK in Rust language

山下 亮輔 † 山崎 憲一 †
Ryosuke Yamashita Kenichi Yamazaki

1. はじめに

システムプログラミングにおいて，C や C++ を用いる場合，Use-After-Free や Double-Free 等からメモリ安全性が脅かされる可能性が高くなる．そこで，メモリ安全な言語を利用することが有用である．Rust 言語は特にコンパイラによってメモリ安全性を静的に保証する．

Rust 言語は新興言語ゆえに，既存ライブラリやプログラムが十分にあるとは言えない．このため，外部関数インターフェイス（Foreign Function Interface, FFI）を利用して C 言語で記述された既存のコードを組み込むことがある．しかし，C 言語等のメモリ非安全な言語を用いると，メモリ安全性が損なわれる可能性がある．

そこで，本研究では不注意によるメモリエラーを低減することを目的とし，外部関数をプロセス内で分離する方法を提案する．ここで，不注意によるメモリエラーとは，意図的ではない C 言語からの改ざんや読み取りを言う．本稿では，悪意ある C 言語のコードについては対象としない．

2. 既存研究

外部関数を分離し安全に実行する手法は大別して 2 種類存在する．

Sandcrust[1] は，マクロによって外部関数を独立したプロセスとして分離する．異なるメモリ空間となるため，保護したいメモリ領域が影響を受けにくく，セキュリティの面で優れている．しかし，分離したプロセスとの IPC とコンテキストスイッチが大きなオーバヘッドとなる．

もう一つの方法であるプロセス内分離は，同一プロセス内で外部関数を分離する方法である．この場合，同じアドレス空間を共有するため，パフォーマンスに優れるがメモリ安全性の確保が課題となる．

プロセス内分離の手法として，Intel CPU の機能である MPK を用いた手法がある．MPK は，ページ単位でメモリ保護を可能にする機能である．MPK が搭載された CPU には，PKRU レジスタが備わっており，このレジスタの値に応じて 16 個の保護キーに対するページグループのアクセスが制御される．ページテーブルに保護キーを紐づけたあとは，ユーザーレベルで PKRU を変更することで，ページグループのアクセス権限の変更が可能となる．これにより，外部関数の実行前に一時的にアクセス権限を変更し，終了後に再度変更にすることで，外部関数内でのバグ等から特定のメモリ領域を低いオーバヘッドで保護することができる．セキュリティ上の欠点として，外部関数側で PKRU の値を意図的に書き換えることは可能であるが，今回はこれについて考慮しない．

†芝浦工業大学大学院, Shibaura Institute Of Technology

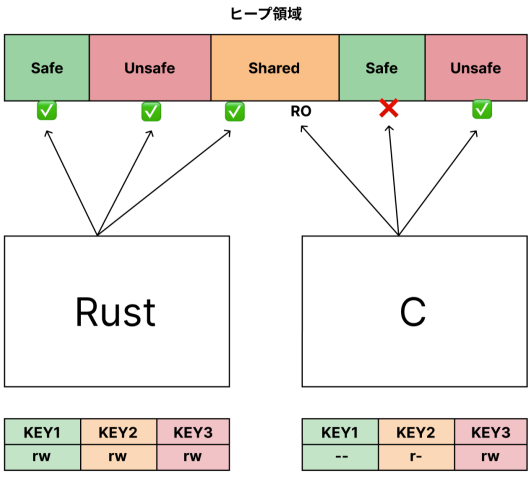

図 1 システム構成図

Galeed[2] は MPK を利用し，外部関数からデータの改ざんを防ぐための研究である．ランタイム時の MPK 制御と疑似ポインタを用いており，外部関数側はメモリアドレスとしてのポインタではなく疑似ポインタを通じて Rust とデータの受け渡しを行う．そのため，ヒープ領域に割り当てられた構造体のフィールドに対するゲッタとセッタを経由してメモリにアクセスする必要があり，オーバヘッドが生じる．

Donky[3] はドメインというセキュリティグループと MPK のハードウェア拡張によってメモリ保護を実現する．ドメインに紐づけられた関数は，ドメイン内で許可されたメモリ領域にアクセス可能となる．しかし，C 言語のみを用いるため，異なるドメイン間でデータを渡したあとに，呼び出し元がそのデータを操作しないことを保証できない．

3. 提案方式

2 種類の分離手法は，いずれの方法においてもオーバヘッドの低減に課題がある．また，C 言語単体を用いる場合，人為的なメモリエラーが発生する可能性が高くなる．そこで本研究では，オーバヘッド低減の課題は MPK を用いることで解決し，人為的なメモリエラーの発生は Rust のデータ型による保護により解決する．

3.1. 基本設計

ヒープ領域をページを基本単位とする 3 つの領域に分割し，各領域に MPK を割り当ててアクセス権限を管理する．具体的には，ヒープ領域を Safe 領域，Unsafe 領域，そして Shared 領域の 3 種類のセグメントに分割する．

Safe 領域は、Rust 側にのみ読み書き可能であり，外部言語からは一切読み書き出来ない領域である．そのため，外部関数

領域	API	権限	
		Rust	外部関数
Safe	Box	rw	- -
Unsafe	UnsafeBox	rw	rw
Shared	SharedBox	rw	r-

図 2 対応図表 (r=読み取り可能，w=書き込み可能)

と共有する必要のない機密情報のデータ保持に適している．

Unsafe 領域は，Rust 側と外部関数の両方において読み書きが可能である．この領域は外部関数が唯一自由に操作できる領域となるため，機密性がない頻繁に書き込みが行われる値の共有に適している．

Shared 領域では，Rust 側は読み書きが可能である一方で，外部関数は読み込みのみが可能である．これにより，外部関数側では読み取り参照のみ行い，データの改ざんを防ぐことができる．そのため，サイズが大きく改ざんを防ぎたい機密情報や読み込みのみ行いたいファイルデータのバッファなどを渡す際に有用である．

3.2. API

Rust には，ヒープメモリ割り当ての処理をカプセル化しプログラマにメモリ割り当てを透過的にする Box 型が備わっている．本提案では，標準クレートの Box 型は，Safe 領域となる．Unsafe 領域と Shared 領域それぞれからメモリ割り当てを行うデータ型として，UnsafeBox, SharedBox を新たに用意する．図 2 に，対応する領域と API，権限を示す．

Listing1 は，本提案の各 Box に実装される独自定義のトレイトである．トレイトとは Java におけるインタフェースに類似し，型に対して特定の関数群の実装を強制する機能である．

as_ptr() と as_mut_ptr() はそれぞれ非可変値のポインタと可変値のポインタを渡す．これらの関数を用いてポインタを渡す際は，所有権は消費されない．一方で，move_ptr() と move_mut_ptr() は所有権を消費しながらそれぞれ非可変値のポインタ，可変値のポインタを渡す．これらの関数を使い分けることで，ポインタを外部関数に渡した後の Rust 側での関与の有無をプログラム上で静的に記述できる．

Listing2 は，読み取り専用のデータから，書き込み可能なデータを更新するサンプルコードである．分離したい外部関数を呼ぶ際は，isolate という独自定義の特殊構文で実行する．isolate 内での外部関数は，制限されたアクセス権限で実行される．

main 関数のなかではデータサイズが大きく，改ざんを防ぎたいデータを SharedBox で割り当て，外部関数側で書き込みを許可する値を UnsafeBox で割り当てている．CustomBox トレイトのメソッドを用いることで，外部関数の引数の型と，所有権の受け渡しに応じてポインタの渡し方を変更できる．

なお，C 側のグローバル変数や malloc で確保したメモリは Unsafe 領域に割り当てられる．そのため，外部関数がポインタを戻り値として返すとき，Rust 側は受け取ったポインタを UnsafeBox にデータ型変換して用いる．

```
trait CustomBox<T: ?Sized> {
  unsafe fn as_ptr(&self) -> *const T;
  unsafe fn as_mut_ptr(&mut self) -> *mut T;
  unsafe fn move_ptr(self) -> *const T;
  unsafe fn move_mut_ptr(self) -> *mut T;
}
```

Listing 1 カスタムトレイト

```
extern "C" {
  fn do_stuff(person: *const Person, health_info: *mut HealthInfo);
}

#[repr(C)]
struct Person {
  weight: f64,
  height: f64,
  ..., // その他個人情報
}

#[repr(C)]
struct HealthInfo {
  bmi: f64,
  ..., // その他健康情報
}

fn main() {
  /* 個人情報の取得 */
  let person= Person::new( 個人情報 );

  /* Shared 領域に割り当てる */
  let read_only_data = SharedBox::new(person);

  /* 初期値を Unsafe 領域に割り当てる */
  let mut writable_data = UnsafeBox::new(HealthInfo::new());

  /* 個人情報から健康情報を取得 */
  isolate!(
    do_stuff,read_only_data.move_ptr(),writable_data.as_ptr_mut());
  );

  /* read_only_data の所有権は移動し，以降触れられない */
}
```

Listing 2 サンプルコード

4. 実装

Rust には，Box でヒープ割り当てを行うグローバルアロケータをユーザプログラムで実装できる機能がある．この機能を利用し，Safe 領域，Shared 領域，Unsafe 領域にそれぞれ独自のアロケータを用意し，Box，SharedBox, UnsafeBox を実現する．また，プログラマが利用する MPK 及びアロケータの初期化関数を実装する．外部関数を呼ぶ前後で MPK の変更を行う isolate 構文も実装する．

5. まとめ

本研究では，Rust 言語における MPK を用いた外部関数呼び出しのプロセス内分離の方法を提案した．今後は実装および評価を行う．

参考文献

[1] B. Lamowski, C. Weinhold, A. Lackorzynski, and H. Härtig, “Sandcrust: Automatic sandboxing of unsafe components in rust,” in *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, pp. 51–57, 2017.

[2] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow, “Keeping safe rust safe with galeed,” in *Proceedings of the 37th Annual Computer Security Applications Conference*, pp. 824–836, 2021.

[3] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, “Donky: Domain keys–efficient {In-Process} isolation for {RISC-V} and x86,” in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1677–1694, 2020.