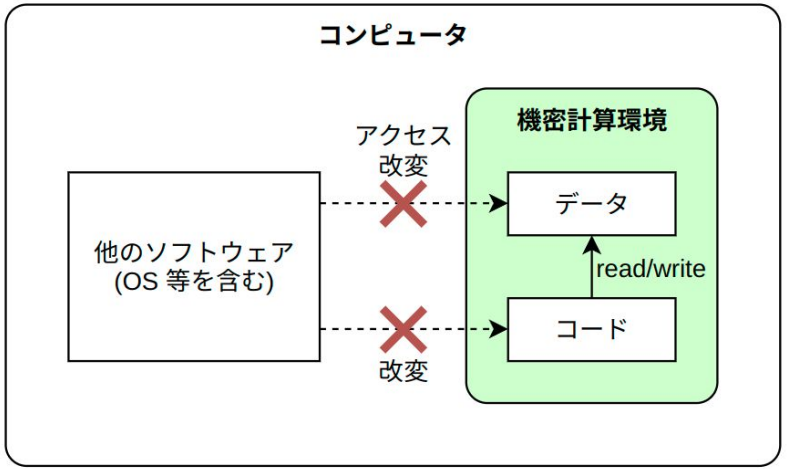
Intel SGX enclave で動く Ruby インタプリタ

山地 博之（筑波大学）　　新城 靖（筑波大学）

機密計算環境

- データを秘匿しながらプログラムを実行したい
  - **クラウド：**
    信頼できない事業者が提供する環境で、
    機密処理を行う
    - Microsoft Azure Confidential Computing
    - Google Cloud Confidential Computing
  - **PC：**
    - サーバの機密処理をオフローディング
    - 著作権保護
- コードの**完全性**、データの**機密性**と**完全性**



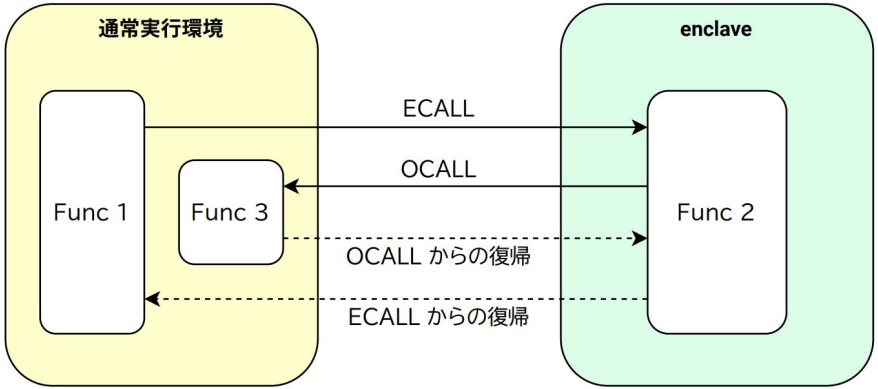
機密計算環境の代表的な実装

- Arm TrustZone
  - Normal World で動くプロセスが、Secure World の OS で機密処理のプロセスを実行
- AMD SEV（Secure Encrypted Virtualization）
  - VM 全体を暗号化
- Intel SGX（Software Guard Extensions）
  - 1 プロセス内で**関数単位（ページ単位）**で暗号化できる
  - 環境内で実行するプログラムが小さい → セキュリティ侵害の可能性を最小限にできる

本研究では SGX を使用する

Intel SGX（Software Guard Extensions）

- 機密計算環境を提供する Intel アーキテクチャの拡張
  - 第 6 世代以降の Intel Core プロセッサに搭載
- メモリ上に enclave と呼ばれる保護領域を作成
  - OS やハイパーバイザを含む他の全てのプログラムから隔離
- 関数単位で保護する
- 環境間の境界を跨ぐ関数呼び出しの仕組みを提供
  - ECALL:
    enclave の外から内への関数呼び出し
  - OCALL:
    enclave の内から外への関数呼び出し
- enclave でシステムコールが使えない



SGX とプログラム開発

- C, C++, Rust, Go 向けの開発キットが存在
  - enclave 向けに機能が制限された標準ライブラリを提供
  - コンパイル型言語
- Ruby で開発したい
  - 高水準な言語機能を利用できる
    - ガベージコレクション（GC）
    - 文字列処理
  - C, C++ と比較して、メモリ破壊のリスクがない

enclave 内でインタプリタを実行する方法が考えられる

SGX の制約とインタプリタ

enclave では **システムコールが使えない**

- ScriptShield [1]
  - libc の改変によって、OCALL による透過的なシステムコールを実現
  - Lua, JavaScript, Squirrel の実行が可能
  - **問題点**：
    システムコールの実装が不完全で、それらを使うインタプリタに非対応
- WebAssembly（Wasm）の活用
  - ruby.wasm + enclave で動く Wasm ランタイム [2]
  - **問題点**：
    ランタイムの実装が不完全でインタプリタが起動しない
    動作したとしても、Wasm が提供しないスレッドなどの機能は非対応

[1] Huibo Wang, Erick Bauman, Vishal Karande, et al. Running Language Interpreters Inside SGX: A Lightweight, Legacy-Compatible Script Code Hardening Approach. In Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS '19), pp. 114–121, 2019.
[2] WebAssembly Micro Runtime (WAMR). https://github.com/bytecodealliance/wasm-micro-runtime. Accessed: 2023-11-08.

目標

SGX enclave で CRuby を動かす

SGX のプログラム開発に対して、Ruby によって実現される以下を提供

- 高水準な言語機能
- メモリ破壊のリスクを回避

方針

CRuby の新たなビルド対象環境として SGX を定義する

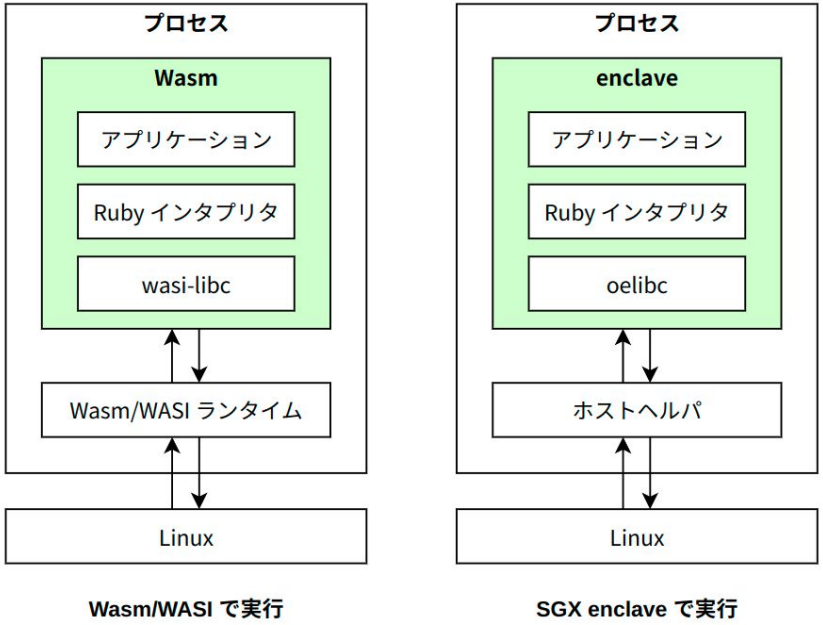
- enclave の外の Linux 環境とは異なる標準ライブラリの API を提供
- Ruby インタプリタ本体が使う API は全て提供する
- アプリケーションがよく利用する API は提供する
- untrusted なものも提供するが、アプリケーション開発者の判断で利用する
  - 例: `clock_gettime`
- セマンティクス的に不可能なものは提供しない
  - 例: `getrusage`, `getrlimit`, `mprotect`

Wasm/WASI と SGX enclave

- 構造が類似している
- ruby.wasm は wasi-libc を使用
  - musl libc のサブセット

SGX enclave 対応においては、

- Open Enclave SDK が提供する oelibc を使用する
  - 同じく musl libc のサブセットで、提供される機能が類似
- インタプリタの移植は、ruby.wasm を参考にする



configure

- Linux 向けの Ruby インタプリタは、システムコールを多用する
  - システムコールは enclave で実行できない
  - セマンティクスが変わるものは、OCALL では対応できない
    - `getrusage`, `getrlimit`, `mprotect`

- Ruby インタプリタは、configure スクリプトにより様々な環境に対応する
  - 環境に応じた Makefile やマクロを生成し、使用するライブラリ関数を切り替える

configure スクリプトのビルド環境に SGX を追加する

- `LIBS`, `LDFLAGS`, `CFLAGS`, `CPPFLAGS` の設定
- enclave で利用できない機能の無効化

__sgx__ マクロによる分岐

- configure で対応できない箇所は __sgx__ マクロで対応
  - CFLAGS に -D__sgx__ を付ける
  - 使用できないライブラリ関数の呼び出しをスキップする

```
#if defined(RUSAGE_SELF) && !defined(__sgx__)
        {
            struct rusage usage;
            if (getrusage(RUSAGE_SELF, &usage) == 0) {
                省略
```

GC とスレッドのスタック

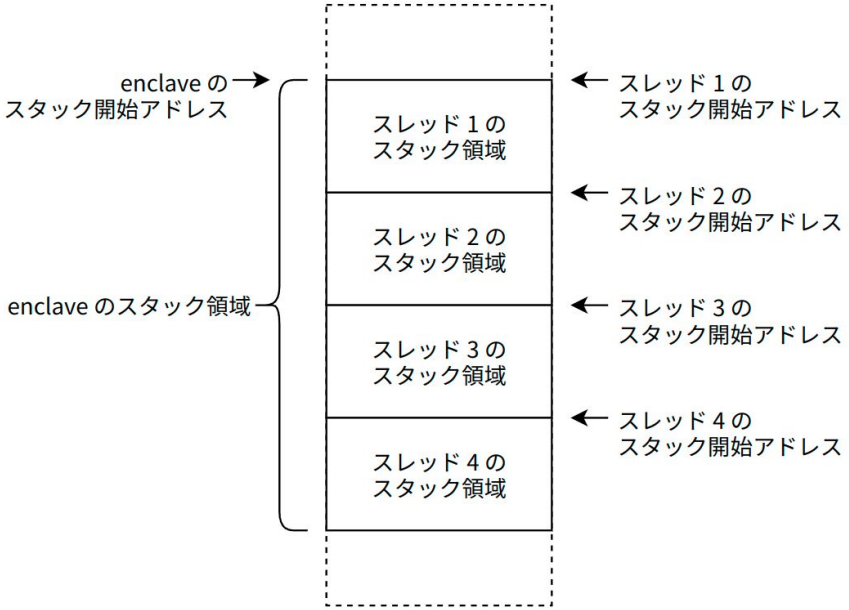
- Ruby GC
  - スタックやレジスタを走査し、参照のあるオブジェクトをマークする
- スタックの走査で使用するスタックの開始アドレスは、スレッド生成時に取得しておく
- Linux では、`pthread_attr_getstack` を使っている

- **問題:** enclave で使える libc が `pthread_attr_getstack` を提供しない
  - セマンティクスが変わるため OCALL で実装不可能

GC とスレッドのスタック

- SGX では、enclave 初期化時に以下を静的に設定する
  - enclave が利用できる最大スタックサイズ
  - ECALL で enclave に入れる最大スレッド数
- スタック全体を最大スレッド数で分割し、各スレッドのスタックとして使用
- `pthread_attr_getstack` を独自に実装
  - `oe_sgx_get_td` および `td_to_tcs` を使って各スレッドのスタック開始アドレス取得

    td : thread data
    tcs : thread control structure



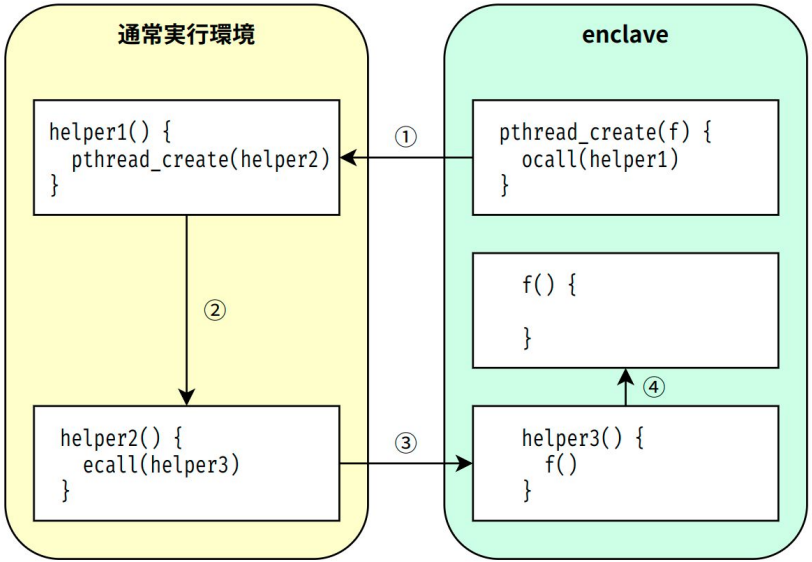
スレッド API

- Linux 向け Ruby インタプリタは、Pthread によってスレッド API を実装している
- oelibc は、Pthread API を部分的に提供している
  - mutex、条件変数： enclave 内で動作する独自の実装

- **問題:** enclave で使える Pthread API は、スレッド生成機能を提供しない
  - enclave 外の Pthread のコードをそのまま持ってきても動作しない

スレッド生成機能の実装

- enclave 内の各スレッドについて、enclave 外でスレッドを作る
- 以下のように実装
  1. OCALL で enclave の外に出る
  2. enclave の外でスレッドを生成する
  3. 生成したスレッドから ECALL で enclave 内の関数を呼ぶ
  4. 別スレッドで実行する関数を呼ぶ
- 元のスレッドは OCALL からの復帰で enclave に戻る



detach されたスレッドの生成

- Ruby インタプリタは detach されたスレッドを生成する
- **問題:** oelibc は `pthread_attr_setdetachstate` を提供しない

- `pthread_create` で与えられた属性 `detachstate` を、OCALL 時にホストに渡す
- ホストは受け取った `detachstate` を反映してスレッドを生成する

条件変数によるタイムアウト付きブロック

- oelibc は `pthread_cond_wait` を提供している
  - スピンロックは enclave 内で行う
  - スレッドのスリープ時は futex システムコールを OCALL で実行する

```
syscall(__NR_futex, 省略, FUTEX_WAIT_PRIVATE, NULL, 0);
```

- **問題:** `pthread_cond_timedwait` が不足している
  - `pthread_cond_wait` にタイムアウト機能を付けたもの
  - ホストの `pthread_cond_timedwait` は利用できない

- `pthread_cond_wait` の実装を参考に、`pthread_cond_timedwait` を実装した
  - スピンロックの方法は同じ
  - futex スリープ時にタイムアウトを指定する

```
syscall(__NR_futex, 省略, FUTEX_WAIT_BITSET_PRIVATE | FUTEX_CLOCK_REALTIME, timeout, FUTEX_BITSET_MATCH_ANY);
```

現在動く Ruby コード

- 単純な計算
- GC
- スレッド API
- ファイル IO

残された課題

- 暗号化されたファイル IO
- リモートアテステーション
  - enclave のコードの完全性を遠隔から検証する
- 通常環境で動作するインタプリタとの連携実行
  - Ruby プログラムの一部のみを enclave で実行可能に

まとめ

目標

- SGX enclave で CRuby を動かす

方針

- 新たなビルド環境として SGX を定義する
- スレッドのスタックアドレスの取得を実装する
- スレッド生成機能を実装する

成果

- GC やスレッド API が動作する Ruby インタプリタを enclave で実行し、enclave 内で Ruby コードを実行可能になった