3. SGXプログラミングの基礎

Ao Sakurai

2025年度セキュリティキャンプ全国大会
L3 – TEEビルド&スクラップゼミ

本セクションの目標

- SGXプログラミングの基本中の基本について解説する

- 実際にEnclaveを起動して呼び出し、Enclave内において
  ごく簡単なコードを実行する、「Hello, Enclave」を実装する

必携書

- SGXプログラミングを行う上では、以下の開発者リファレンスが間違いなく必須となる
  https://download.01.org/intel-sgx/sgx-linux/2.26/docs/Intel_SGX_Developer_Reference_Linux_2.26_Open_Source.pdf

- いずれ118ページくらいまでは全て読んでおいた方が良いが、本ゼミではスライドでの解説でほぼ網羅するようにはしている

SGXプログラミングにおける制約

SGXプログラミングに伴う制約

- 非常に攻撃力の高い脅威モデルにも対応できるSGXであるが、**その代償**は**苛烈な負担**という形で**開発者に降りかかる**

- 加えて、Enclave**境界を何度も跨ぎながら**の**独特のコード開発**が必要となるため、SGXプログラミングは**難易度が非常に高い**

- SGXElide[1]という技術の論文では、元々**412**行のコードをSGX上で動くようにした所、**3523**行にまで肥大化したと報告している
  - やってみると分かるが**誇張でも何でもない**

EPCサイズ上限に伴う制約（1/4）

- SGX、ひいてはTEEは、**TCB**の**サイズ**を**可能な限り小さくする**事を**根底の思想**として持っている

- 開発者リファレンスにも書いてある通り、TCB内の**コード**が小さければ**潜在的な脆弱性も根本的に少なくなり**、TCB内の**データ**が小さければ**いざ漏洩しても被害を最小限に抑えられる**という考え方らしい
  - OSを含むVMをまるごとTCBに収める、AMD SEVのようなVM型のTEEが本来は異端である旨が窺える

EPCサイズ上限に伴う制約（2/4）

- SGXも本来のTEEの例に漏れずTCBの軽量化を開発者に要求している

- 前のセクションで説明した通り、ユーザが自由に使用できるEPC（≒TCB）サイズは**本来は**たったの96MB
  - コードとデータ両方込みで96MB
  - BIOS設定やモデルによってはさらに小さい場合もある
  - 何でこのサイズなのか？→**Intelの独断[2]**
    - 一応、MEEが完全性保持のために維持するマークルツリーのサイズの限界に近いためという擁護はできる[8][9]

EPCサイズ上限に伴う制約（3/4）

- ただし、Linuxドライバの場合EPCの**動的ページング**（EWB命令やELD系命令によるページスワップ）**が可能**であるため、**ページスワップで対応できる範囲**であれば、擬似的にEPCサイズをランタイム中にて**自動的に拡張**できる

- かつ、Scalable-SGXではEPCサイズを1ソケットあたり最大**512GB**まで拡張できる [3]
  - 前のセクションで述べた通り、MEEではなくTME-MKを用いているため

EPCサイズ上限に伴う制約（4/4）

- EPCメモリをある程度拡張できるとしても、実装の際には引き続き**可能な限りTCBサイズ**（特にTCB内の**データサイズ**）を**抑える設計を心がけるべきである**

- 例えば、脆弱性を突いてキャッシュ階層からEnclave秘密情報を抽出する**Foreshadow**攻撃や**ÆPIC Leak**攻撃では、Enclave内の**任意のデータをキャッシュ階層にロード**させる「**Enclave Shaking**」という技術を使用している
  - そもそもEnclave内に無ければ防げるので、TCBサイズは小さい方が無難
  - 攻撃の詳細は後のセッションで解説

SGX2（1/3）

- Enclave初期化後にも動的にEPCサイズを変更する事を可能にするSGXとして、**SGX2**というものが開発された

- SGX2では、**EAUG**というENCLSのリーフ関数により、Enclave初期化後でもEPCページを追加（**EDMM**機能）出来る
  - **元々のEPCサイズ上限**（例えば96MB）を**超えてEPCページを追加**できるもの**ではない**事に注意[7]

- SGX2では、その他RDTSC命令のような追加の命令をEnclave内から利用できる機能も提供している
  - **RDTSC**：クロック周波数ベースの信頼可能な時間ソースから時間情報を取得する命令[12]

SGX2（2/3）

- が、**SGX2対応の量販マシンは何とこれしかない**[4]

Device	Vendor	Model	Source	Date	Confirmed
Mini PC	Intel NUC Kit	NUC7CJYH, NUC7PJYH	Issue 48, Pull Request 68	4 Apr 2019	NUC7CJYH, NUC7PJYH
Laptop	Dell	XPS 13 9300	Issue 75	24 Feb 2021	XPS 13 9300
Laptop	Lenovo	Ideapad Yoga C940	Issue 77	13 Mar 2021	Ideapad Yoga C940
Server	SuperMicro	X12SPM-TF	PR 87	18 Jan 2022	SuperMicro X12SPM-TF with Xeon Gold 5315Y

- その他、明確にSGX2に対応しているクラウドサービスは現状Alibaba Cloudのみである模様[4]
  - コマンドで確かめると、Azureも実は対応している模様

SGX2（3/3）

- ちなみに、**SGX2**は**Scalable-SGX**とは全く**無関係の概念**である

- 稀に**レガシーSGX**を**SGX1**、**Scalable-SGX**を**SGX2**と表現している文献が存在するが、**これは誤り**である
  - 国際論文レベルでもこの誤用をしている論文がある[11][12]

- 「**EPCページを増やせる**」という字面が共通している事や、**SGXのドキュメント**が**カオス**である事から生まれた**誤解**であると考えられる

脅威モデルに起因する制約

- 前のセクションで解説した通り、**SGXのEnclave**は**OSですら信頼不可能**であると見なす**脅威モデル**を想定している

- これは裏を返せば「**Enclave内**では**OSに直接依存する処理**は**一切使用する事が出来ない**」という**苛烈な制約**を意味する

Enclave内で使用できないコード（1/2）

- **システムコール（OSカーネル**の機能の要求）**を発行する関数**は **全て使用できない**
  - 例：**printf**, **scanf**, exit, fopen, fork, exec, time, setlocale

- メモリ破壊を引き起こすような**潜在的に危険な関数**も使用できない
  - 例：strcpy

- 厳密には、Enclave内で使用可能なC/C++ライブラリである専用の **tlibc/tlibcxx**に従わなければならない

Enclave内で使用できないコード（2/2）

- Enclave内のコードに**共有ライブラリ**（*.so）を**ロードする事は出来ない**
  - 例えばOpenSSLやMySQL ConnectorをEnclave内で使用する事はまず不可能

- **静的ライブラリ**（*.a）に変換し、かつその**内容が一切OSの機能に依存しなければ**使えるらしいが、極めて面倒な上に制約も多く、**まず現実的ではない**

- 救済措置的な機能として、Enclave**内**からEnclave**外**の関数を呼び出す**OCALL**という機能がある（詳細は後述）

SGXの動作モード

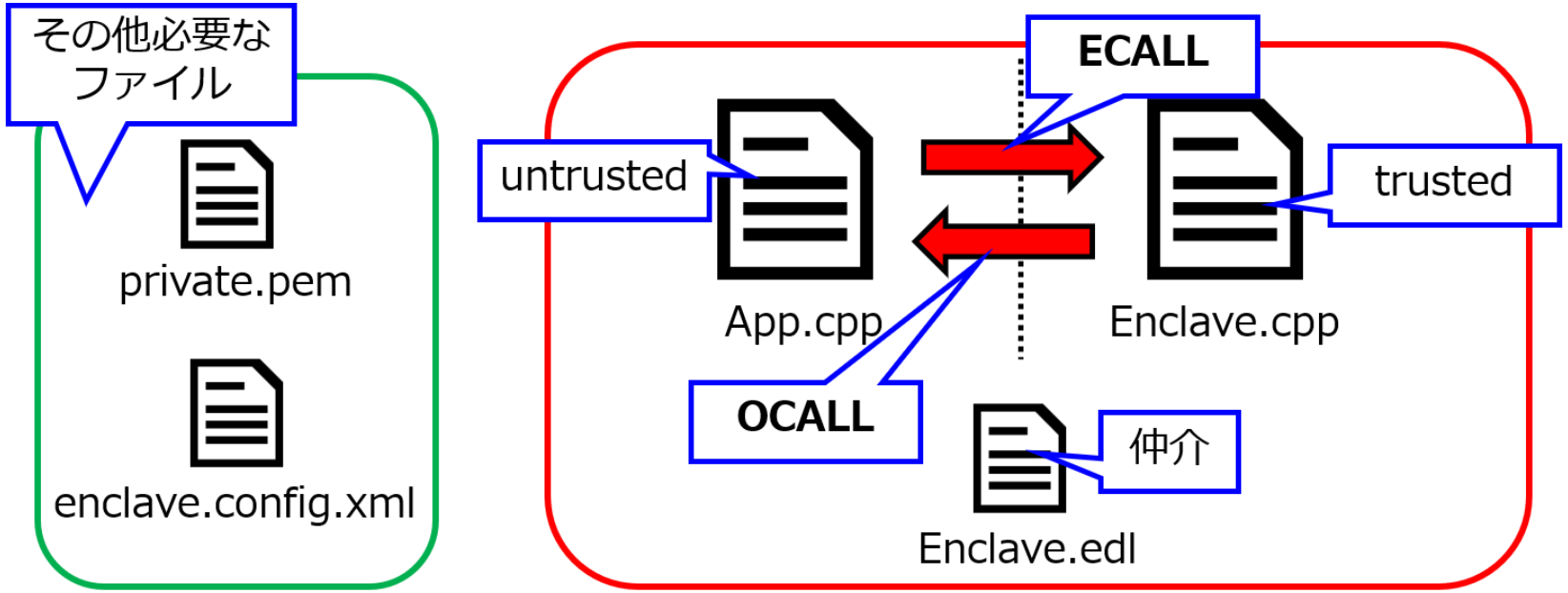
- 本ゼミではSGX対応のAzureインスタンスを用意しているが、**SGX対応のハードウェア環境**を揃えるのは**意外に難しい**
  - 第6～10世代のCoreシリーズ、第2（一部）・3世代Xeon
  - 第11世代以降のCoreシリーズからはSGXは削除されている
  - マザーボードも対応のものにしなければならない

- そこで、SGX対応マシンでなくとも**擬似的**にSGXを動作させられる**シミュレーションモード**（**SIM**モード）が存在する
  - 通常の擬似的でない動作モードは**ハードウェアモード**（**HW**モード）という
  - SIMモードでは、**リモートアテステーション**のような**ハードウェアに強く依存する処理**は**正常に実行する事が出来ない**

SGXプログラミングの流れ

SGXプログラミングにおけるファイル構成（1/2）

- 最低限かつ最小の構成でSGXアプリケーションを開発する場合、概ね以下の図のような構成となる
  - ファイル名はもちろん任意で変えて良い。その場合はMakefileやビルドコマンドも適宜変更する



SGXプログラミングにおけるファイル構成（2/2）

- 前ページのファイルそれぞれの役割は以下の通り：

ファイル名	役割
App.cpp	Enclave**外**の**Untrusted**なプログラムのソースコード
Enclave.cpp	Enclave**内**の**Trusted**なプログラムのソースコード
Enclave.edl	Enclave**内外を跨ぐ**関数のための、**専用言語**で記述する定義ファイル
Enclave.config.xml	Enclaveの**コンフィグファイル**。バージョンやTCS数（=使用可能スレッド数）、デフォルトのEPCヒープサイズ等を設定する
private.pem	ビルドしたEnclaveイメージ（**Enclave.so**）に**署名を行う**ための秘密鍵。MRSIGNER値に直接関係する

ECALL/OCALL（1/2）

- Untrustedコード（App.cpp）でEnclaveを起動後、**Enclave内のコード**を利用するには、**Enclave境界を跨いだ関数呼び出し**が必要となる

- この境界を跨いだ関数呼び出しとして**ECALL**（Enclave CALL）と**OCALL**（Outside CALL）が存在する

ECALL/OCALL（2/2）

■**ECALL**
前ページの例のように、Enclave**外**（App）からEnclave**内**の関数を呼び出す（**Enclaveに進入する；EENTER**）ような関数呼び出し。
SGXの恩恵に預かるのであれば必ず実行する必要がある

■**OCALL**
Enclave**内**から一時的にEnclave**外**に実行を移し（**EEXIT**）、**Enclave外の関数を呼び出す**ような関数呼び出し。
実行後には再度EENTER命令を発行してEnclaveに戻る[5]。
前述の通り、Enclave内の**苛烈なライブラリ制限を回避**するために使用できる。
OCALL先関数内におけるデータは一切保護されないので注意

Edger8r tool（1/3）

- 実は、ECALLやOCALLの宣言・呼び出しは、**そのまま**（App.cppやEnclave.cppに書くだけ）**ではコンパイラが解釈できない**
  - 厳密には、Enclave境界を跨ぐ関数のインタフェースは厳重に管理されなければならない（名前衝突の未然の防止等）ため、通常の関数定義では不十分であるのが理由

- これを橋渡しして解決してくれるツールとして、SGXSDKによって**Edger8r tool**というものが用意されている
  - Edger8rの読み方は「**エジャレーター**」[6]

- このEdger8r toolにECALLやOCALLに関する定義を教えてやるファイルが前述の**Enclave.edl**

Edger8r tool（2/3）

- Enclave.edl内における定義は、その拡張子が示す通り**EDL**（Enclave Definition Language）という、**C言語まがい**の**特殊な専用言語**で記述する必要がある

- EDLに基づき、Edger8rは**エッジ関数**（Edge Routine）と呼ばれる、**極めて厳格に定義**された**ECALL/OCALL呼び出し処理**が記述された、**ソースコード**及び**ヘッダ**を**自動生成**する
  - 後述の通り、Edger8rによって生成されたヘッダは所定の規則でEnclave内外のコードにincludeする必要がある

Edger8r tool（3/3）

- エッジ関数は**基本的に人間が読んで良いような代物ではない怪文書**であり、**実際に読む機会もまずない**

```
static sgx_status_t SGX_CDECL sgx_ecall_test(void* pms)
{
  CHECK_REF_POINTER(pms, sizeof(ms_ecall_test_t));
  //
  // fence after pointer checks
  //
  sgx_lfence();
  ms_ecall_test_t* ms = SGX_CAST(ms_ecall_test_t*, pms);
  sgx_status_t status = SGX_SUCCESS;
  const char* _tmp_message = ms->ms_message;
  size_t _tmp_message_len = ms->ms_message_len;
  size_t _len_message = _tmp_message_len;
  char* _in_message = NULL;

  CHECK_UNIQUE_POINTER(_tmp_message, _len_message);

  //
  // fence after pointer checks
  //
  sgx_lfence();

  if (_tmp_message != NULL && _len_message != 0) {
     _in_message = (char*)malloc(_len_message);
     if (_in_message == NULL) {
        status = SGX_ERROR_OUT_OF_MEMORY;
        goto err;
  ...(後略)
```

エッジ関数の例

SGXSDKのライブラリ

- SGX関連の処理を行うためには、基本的に**SGXSDKで用意された専用のライブラリ**を**include**する必要がある
  - 例えば、Enclaveを起動するsgx_create_enclave()やデストラクトを行うsgx_destroy_enclave()はsgx_urts.hに含まれる
  - また例えば、sgx_ra_context_tというある専用の型の定義はsgx_key_exchange.hに含まれている

- これらの**SGXAPIや専用の型、ヘッダファイルは非常に数が多い**ため、**適宜開発者リファレンスを参照して使い方を調べる**必要がある
  - たまにIntelクオリティでリファレンスにも載ってないものがあるので、その場合は**SGXSDK内のコードを直接参照する**

Enclave内外のコードのビルド・署名（1/5）

- （エッジ関数含め）一通りコードを揃えたら、今度はそれらの**ビルド**を実施する

- App.cpp（Untrustedコード）に関しては、通常通りコンパイル・ビルドし、**実行バイナリを生成**する

- Enclaveコードに関しては、**Enclaveイメージ**（Enclave.so ; 厳密には**共有ライブラリ**）を生成し、さらにprivate.pemで**署名**を行う（**Enclave.signed.so**の生成）
  - 厳密には**SIGSTRUCT構造体**を生成しEnclaveイメージに組み込む処理

Enclave内外のコードのビルド・署名（2/5）

- ちなみに、生成されるSIGSTRUCT構造体は、署名情報として以下の情報を格納している：
  - RSAモジュラスm（これの256bit SHA-2ハッシュ値がMRSIGNER）
  - RSA暗号化における指数
  - RSA署名s
  - $Q1 = \lfloor \frac{s^2}{m} \rfloor$
  - $Q2 = \left\lfloor \frac{s^3 - Q_1 \times s \times m}{m} \right\rfloor$

Enclave内外のコードのビルド・署名（3/5）

- Enclaveイメージの署名には**sgx_sign**というSGXSDKによって提供されている**署名関係用ツール**を使用する

- 署名に用いる秘密鍵は以下のコマンドで生成できる

```
openssl genrsa -out private_key.pem -3 3072
```

- 署名方式として、**シングルステップ署名**と**2ステップ署名**の2つが存在する

Enclave内外のコードのビルド・署名（4/5）

- **シングルステップ署名**：文字通り**1コマンド**でEnclaveイメージに署名する方式

- **2ステップ署名**：一度**署名用マテリアル**を生成し、それに対し前述の秘密鍵による**署名**を打ち、その後Enclave.soと**結合**させてEnclave.signed.soを生成する方式
  - 署名用マテリアル：そのEnclaveのSIGSTRUCTの”ヘッダ”と”ボディ”[5]
  - 最後の結合により、結果としてSIGSTRUCTの残りの部分（”署名”と”バッファ”）部分が充足される形になる

Enclave内外のコードのビルド・署名（5/5）

- シングルステップ署名の方が楽ではあるが、商用環境では**Enclave署名鍵**を**厳密に管理**する必要がある

- 例えば**ハードウェア・セキュリティ・モジュール**（HSM）で鍵管理と署名を行う場合、**2ステップ署名**であれば**sgx_sign**に**依存せず**、OpenSSL等のツールで**署名**ができる

- よって、開発者リファレンスでも**シングルステップ署名**は**デバッグ版Enclave**向け、**2ステップ署名**は**製品版Enclave**向けであると言及されている

Hello SGX

SGXプログラミングの手順

- SGXプログラミングの**進め方の基本**は**以下の4点**である：
  - Enclaveの作成・起動
  - ECALL関数の定義・ECALLの実行
  - OCALL関数の定義・OCALLの実行（OCALL使用時のみ）
  - EDLの記述

- ビルドコマンドの記述やEdger8r及びsgx_signの呼び出しなどは、本ゼミでは**用意してあるMakefileで完結する**ので気にしなくて良い

- 各SGX用APIや型の使用に必要なインクルードファイルは**開発者リファレンス**で調べる

Edger8rにより生成されるヘッダ

- Edger8rを実行すると、Enclaveコードの**拡張子抜きファイル名**をベースとしたファイル名の**ヘッダが自動生成**される

- Enclave.cppであればEnclave_u.hとEnclave_t.h
  - Enclave_u.hのuはUntrustedの意。**OCALLインタフェースが記載**
  - Enclave_t.hのtはTrustedの意。**ECALLインタフェースが記載**

- App.cppとEnclave.cppはそれぞれ**相手のインタフェース**を知る必要があるので、**App.cppでEnclave_t.h**を、**Enclave.cppでEnclave_u.h**を**インクルード**する必要がある

App.cpp – 起動トークンの準備

- **LEが非推奨化**したのでもはや**形骸化した儀式**でしか無いが、Enclave作成時に**過去に生成された起動トークン**を渡す事が出来る
  - 現在は渡さなくても**製品版Enclave**ですら**FLCで自動的に起動許可**されるので**本当にいらない機能**

- APIの仕様上**sgx_launch_token_t**型の変数を渡さなければならないので、この型の0埋めした適当な変数を用意して終了

```
sgx_launch_token_t token = {0};
```

- App側の記述は普通にmain関数に記述するので良い。必要に応じて特定の処理を関数化するのももちろんOK

App.cpp – Enclaveの作成・起動

- Enclaveの起動に必要な情報を揃えたら、**sgx_create_enclave()**関数でEnclaveの作成・起動を実施する

```
status = sgx_create_enclave(enclave_name.c_str(),
SGX_DEBUG_FLAG, &token, &updated, &global_eid, NULL);
```

- 引数は順に署名済みEnclaveイメージのファイル名、Enclaveのデバッグフラグ（マクロ）、起動トークン（形骸化）、起動トークン更新フラグ（形骸化）、生成したEnclaveのID（ポインタ経由で取得可能）、その他Enclave情報を得るポインタ

Enclave.cpp – ECALL関数の記述

- ECALLで呼び出す関数を**Enclave側**で定義する。今回は受け取った文字列を**OCALLでそのまま標準出力**する~~SGXを使う意味の全く無い~~動作定義とする
  - 戻り値として**適当な整数**も返すようにする

```
#include "Enclave_t.h"
#include <sgx_trts.h>

int ecall_test(const char *message, size_t message_len)
{
   ocall_print(message);
   return 1234;
}
```

App.cpp – OCALL関数の記述

- OCALLで呼び出す関数を**App側**で定義する。これはApp側なので**SGXの制約なしに気ままに書ける**
  - 今回は引数で渡された文字列を標準出力するような動作にする

```
void ocall_print(const char* str)
{
    std::cout << "Output from OCALL: " << std::endl;
    std::cout << str << std::endl;
    return;
}
```

Enclave.edl – EDLファイルの記述（1/4）

- **SGXプログラミング初歩における最難関**。EDLと呼ばれる**独自言語**で**様々な属性**を**厳密に指定**しなければならない

- 今回の場合のEDLは以下のようになる：

```
enclave
{
    trusted
    {
        /*These are ECALL defines.*/
        public int ecall_test([in, size=message_len]const char *message,
            size_t message_len);
    };
    untrusted
    {
        /*These are OCALL defines.*/
        void ocall_print([in, string]const char *str);
    };
};
```

Enclave.edl – EDLファイルの記述（2/4）

- **trusted**ブロックに**ECALLについての定義**を、**untrusted**ブロックに**OCALLについての定義**を**記述**する

- **特定の条件下**では**特定のヘッダ**や**他のEDL**を**インポート**する必要があるが、**この場では不要**

Enclave.edl – EDLファイルの記述（3/4）

- 極まってくると**単一のECALL**のためのEDL定義ですら以下のように**地獄の様相を帯びてくる**

```
public sgx_status_t store_vcf_contexts(sgx_ra_context_t context,
    [in, size=vctx_cipherlen]uint8_t *vctx_cipher,
    size_t vctx_cipherlen, [in, out, size=12]uint8_t *vctx_iv,
    [in, out, size=16]uint8_t *vctx_tag,
    [in, size=ivlen]uint8_t *iv_array, size_t ivlen,
    [in, size=taglen]uint8_t *tag_array, size_t taglen,
    [out, size=emsg_len]uint8_t *error_msg_cipher, size_t emsg_len,
    [out]size_t *emsg_cipher_len);
```

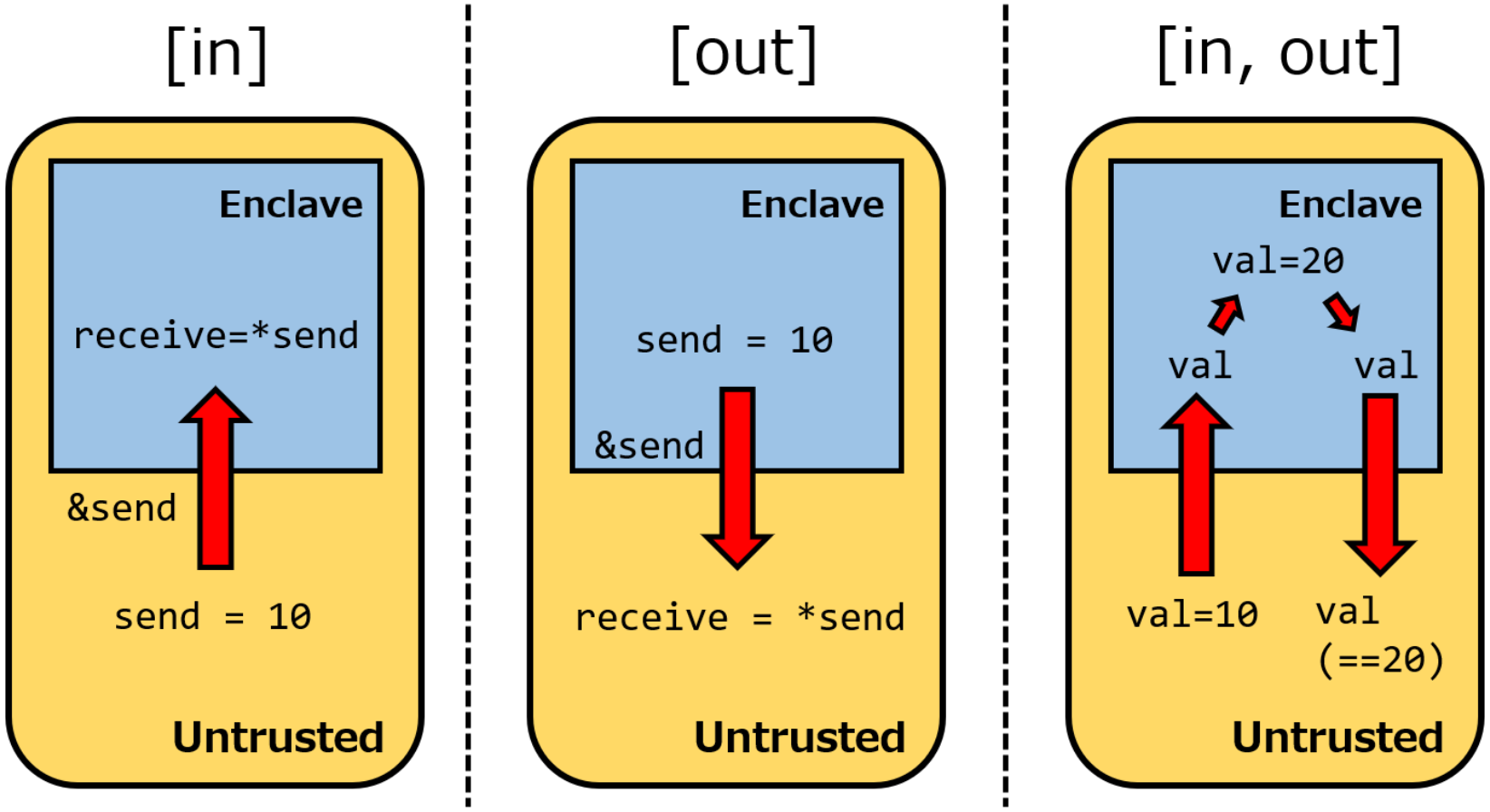
Enclave.edl – EDLファイルの記述（4/4）

- EDLは見ているだけで鬱になりそうかも知れないが、よく見ると**ベース**はC/C++における**関数のプロトタイプ宣言**である事が分かる

- それに加え、以下の要素が**EDL独特**であり、事を複雑にしている：
  - **ECALL定義**では**必ず先頭にpublic修飾子をつける**事
  - **ポインタ型の引数**がある場合、その**方向属性**（とバッファであれば**バッファサイズ**）を**指定しなければならない**事
  - 基本的に**C言語で使える要素**のみが**使用可能**であり、STLやスマートポインタのように**C++依存の要素**は**使用できない**

Enclave.edl – ポインタの方向属性（1/5）

- 方向属性には**[in]**, **[out]**, **[in, out]**, **[user_check]**の4種類が存在する（図はECALLの場合を示している）



Enclave.edl – ポインタの方向属性（2/5）

■ **[in]属性**

- そのポインタを**渡す側**（**呼び出し側**）が、**呼び出し先に値を
コピー**するための属性。「**アップロード**」のイメージに近い
  - ECALLであれば、App側がそのポインタで保持する値を
Enclave側に渡してコピーする事が出来る

- このポインタを介して、**呼び出し先の値**を**呼び出し側に
持ってくる**事は**出来ない（これは[out]の仕事）**

Enclave.edl – ポインタの方向属性（3/5）

■ **[out]属性**

- そのポインタを**渡す先**（**呼び出し先**）の値を、**呼び出し元（渡す側）に持ってくる**ための属性。「**ダウンロード**」のイメージに近い
  - ECALLであれば、Enclave側が保持する値をそのポインタ経由でApp側に持ってくる事が出来る
- このポインタを介して、**呼び出し元の値**を**呼び出し先にコピーする**事は**出来ない（これは[in]の仕事）**
- ポインタ（バッファ）を渡してそこにコピーさせる仕組みなので、**呼び出し側で必要サイズ分のバッファを確保**しておく必要がある
  - NULLだったりサイズ不足だったりしてはならない

Enclave.edl – ポインタの方向属性（4/5）

■ **[in, out]属性**

- 文字通り**[in]と[out]双方の性質を併せ持つ**属性で、**呼び出し元の値を呼び出し先にコピー**する事も、**呼び出し先の値を呼び出し元に持ってくる**事も**出来る**属性

- **一見便利そうに見える**が、後述の通り**バッファサイズにも厳密な指定が必要**な事から、**バッファに対しては使いにくい**
  - 何らかの値を引数として投げて、その同一の引数の変数で向こうから持ってくるという実装もあまり美しくないので、**想像以上に使用機会は少ない**

Enclave.edl – ポインタの方向属性（5/5）

■ **[user_check]属性**

- 前述の[in], [out], [in, out]のような**制限が一切入らない属性**

- **バッファサイズ指定も無い**ため、**潜在的なメモリ破壊の可能性**もあり、**危険性が高い**
  - 言い換えれば**フェイルセーフが存在しない**

- 制限がないと言いながら、渡したはずなのに渡ってないといった**直感に反する挙動を示す事も多い**ため、**限りなく推奨しない**
  - あくまでも**最終手段**

Enclave.edl – バッファサイズ属性（1/3）

- 渡すポインタが**バッファを指すポインタ**である場合、**受け渡すサイズ**についても**別個指定しなければならない**
  - 反対に、例えばint a = 0;へのポインタ*aのように、**バッファでない変数へのポインタであれば不要**

- バッファサイズに関連する属性として**[size]**, **[count]**, **[string]**が存在する

- [size]や[count]では、同時に渡している非ポインタな整数型を用いる事も出来る（例：前述のEDLの[in, size=message_len]）

- いずれも**[user_check]属性では使用できない**

Enclave.edl – バッファサイズ属性（2/3）

■ **[size]属性・[count]属性**

- いずれも**直接バッファサイズを指定**する属性
- [in, size=4, count=8]int *buf,のように使う
- [size]と[count]を**両方指定**した場合、バッファサイズは**[size]*[count]**により算出され決定される
- [size]を指定しなかった場合は、[size]の値は**暗黙にその****ポインタが指す型のバイトサイズ**（例：intであれば4）となる
- [count]を指定しなかった場合は、[count]の値は**暗黙に1**となる

Enclave.edl – バッファサイズ属性（3/3）

■ **[string]属性**

- 受け渡すバッファが**char型のバッファ**（**uint8_t型バッファは不可**）であり、かつ**バッファがヌル終端されている**場合にのみ使用可能

- **この条件を満たせば**、[in, string]のようにするだけで文字列を渡せるため便利

- ただし**[in], [in, out]属性でしか使用できない**（**[out]は不可**）
  - その上、Enclaveには**暗号データをuint8_t型バッファで渡す事が非常に多い**ため、この属性を活用できる機会は**驚くほどに少ない**

App.cpp – ECALL関数の呼び出し（1/2）

- 後は以下のようにApp側でECALLにてEnclave内の関数を呼び出すだけ

```
sgx_status_t status = ecall_test(eid, &retval, message, message_len);
```

- …**eidと&retvalはどこから出てきた**？
  - あと**戻り値のsgx_status_tは何**？
  - 元々の関数宣言は

    int ecall_test(const char *message, size_t message_len)
    であるはず

App.cpp – ECALL関数の呼び出し（2/2）

- この**奇怪すぎる現象**は、Edger8rが
  - **戻り値**を**強制的にsgx_status_tに書き換え**、
  - **第1引数**で**呼び出すEnclaveのID**である**sgx_enclave_id_t**を渡し、
  - **第2引数**で**本来の戻り値を受け取るポインタ**を指定する

  ように**改変するという仕様**によって**引き起こされている**
- ECALL関数の**戻り値の型がvoid**である場合は、**第2引数**に**戻り値取得用のポインタ**を**用意する必要がなくなる**
  - 今回の例であれば単純に**&retvalのみがなくなる**
  - Enclave内からEnclave内の別の関数を呼び出す場合は、Edger8rは絡まないので通常通り
  - OCALLでも戻り値がある場合は引数に挿入される。勿論EnclaveIDは不要

Enclave.cpp – OCALL関数の呼び出し

- わずかな差分（Enclave IDの有無）を除き、基本的にECALLと同様であるため、詳細な説明は省略

```
sgx_status_t status = ocall_print(message);
```

- ECALL関数にも共通するが、引数として**変数名がstatus**であるような**sgx_status_t型変数**を**使用してはならない**
  - Edger8rがこの名前と型の変数を**内部で自動生成して使用**するため、**名前衝突**による**コンパイルエラー**が発生してしまう

App.cpp – Enclaveのデストラクト

- 一通りやりたいことが完了したら、sgx_destroy_enclave()で**Enclaveをデストラクト**してから終了する

```
sgx_destroy_enclave(eid);
return 0;
```

ビルドして実行

- makeコマンドでビルド後、./appでUntrustedアプリケーションを実行すると、ECALL込みで一連の処理が行われる

- 以下のような感じの出力が出れば、Enclaveに文字列を渡し、そこからOCALLで標準出力をしており、無事**基本的なSGXプログラミングに成功している**

```
Execute ECALL.

Output from OCALL:
Hello Enclave.
================================================================================
SGX_SUCCESS
Exited SGX function successfully.
================================================================================

Returned integer from ECALL is: 1234

Whole operations have been executed correctly.
```

（参考）Switchless Call（1/4）

- **ECALL**及び**OCALL**には**ある程度のオーバヘッド**が発生するが、これを**大幅に軽減する機能**として**Switchless Call**が用意されている

- Switchless callは、**スレッドの力**を借りる事でECALL/OCALLによるオーバヘッドを**大幅に軽減**する技術
  - 技術的な詳細に関しては本ゼミでは省略

（参考）Switchless Call（2/4）

- Switchless callのためには、Enclave起動のために以下のような**特別な記述**が追加で必要となる
  - Enclaveの起動に**sgx_create_enclave()**ではなく**sgx_create_enclave_ex()**を用いる
  - sgx_create_enclave_ex()よりも前に出てきている諸々を使用するには**sgx_uswitchless.h**をインクルードする

```
#include <sgx_uswitchless.h>

...

sgx_uswitchless_config_t us_config = SGX_USWITCHLESS_CONFIG_INITIALIZER;
void* enclave_ex_p[32] = {0};

enclave_ex_p[SGX_CREATE_ENCLAVE_EX_SWITCHLESS_BIT_IDX] = &us_config;

status = sgx_create_enclave_ex(enclave_image_name.c_str(), SGX_DEBUG_FLAG,
        &token, &updated, &eid, NULL, SGX_CREATE_ENCLAVE_EX_SWITCHLESS,
          (const void**)enclave_ex_p);
```

（参考）Switchless Call（3/4）

- EDLにおいては、Swtichless Callを行いたい関数の定義の末尾に**transition_using_threads**と追記する
- また、SGXSDKにより用意されている**sgx_tswitchless.edl**を**インポート**する

```
enclave
{
    from "sgx_tswitchless.edl" import *;

    trusted
    {
        /*These are ECALL defines.*/
        public int ecall_test([in, size=message_len]const char *message,
            size_t message_len) transition_using_threads;
    };
    untrusted
    {
        /*These are OCALL defines.*/
        void ocall_print([in, string]const char *str)
            transition_using_threads;
    };
};
```

（参考）Switchless Call（4/4）

- ビルドのリンクフラグについては、App側で**lsgx_uswitchless**、Enclave側で**lsgx_tswitchless**を付与する
  - **App（Untrusted）側**

```
App_Link_Flags := $(SGX_COMMON_CFLAGS) -L$(SGX_LIBRARY_PATH) \
                  -Wl,--whole-archive -lsgx_uswitchless -Wl,--no-whole-archive \
                  -lsgx_ukey_exchange \
                  -l$(Urts_Library_Name) -lpthread -lcrypto -lssl
```

  - **Enclave（Trusted）側**
    - --whole-archiveで囲む必要がある

```
Enclave_Link_Flags := $(SGX_COMMON_CFLAGS) -Wl,--no-undefined -nostdlib -nodefaultlibs -nostartfiles -L$(SGX_LIBRARY_PATH) \
                  -Wl,--whole-archive -l$(Trts_Library_Name) -lsgx_tswitchless -Wl,--no-whole-archive \
                  -Wl,--start-group -lsgx_tstdc -lsgx_tcxx -lsgx_tkey_exchange -l$(Crypto_Library_Name) -l$(Service_Library_Name) -Wl,--
end-group \
                  -Wl,-Bstatic -Wl,-Bsymbolic -Wl,--no-undefined \
                  -Wl,-pie,-eenclave_entry -Wl,--export-dynamic  \
                  -Wl,--defsym,__ImageBase=0
```

本セクションのまとめ

- SGX、特にEnclaveプログラムの実装に伴う様々な苛烈な制約について解説を行った

- Enclaveを利用したごく簡単なプログラムの実装を実践する事で、SGXプログラミングに伴う独特な難しさを体験した

参考文献（1/2）

[1]”SgxElide: Enabling Enclave Code Secrecy via Self-Modification”, Erick Bauman et al., https://dl.acm.org/doi/pdf/10.1145/3168833

[2]”Size limitation for EPC in SGX – Intel Community”, 2023/6/6閲覧, https://community.intel.com/t5/Intel-Software-Guard-Extensions/Size-limitation-for-EPC-in-SGX/td-p/1130830?profile.language=ja

[3]”インテル® Xeon® Platinum 8380 プロセッサー”, 2023/6/6閲覧, https://ark.intel.com/content/www/jp/ja/ark/products/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz.html

[4]”SGX-hardware list”, ayeks, https://github.com/ayeks/SGX-hardware

[5]”Intel® Software Guard Extensions (Intel® SGX) SDK for Linux* OS”, Intel, https://download.01.org/intel-sgx/latest/linux-latest/docs/Intel_SGX_Developer_Reference_Linux_2.26_Open_Source.pdf

[6]”Video Series: Intel® Software Guard Extensions—Part 4: Introduction: Enclave Definition Language”, Intel, https://www.intel.com/content/www/us/en/developer/videos/introduction-to-the-enclave-definition-language-intel-sgx.html

[7]”Intel® Software Guard Extensions (Intel® SGX) SGX2”, Intel, Frank McKeen et al., https://caslab.csl.yale.edu/workshops/hasp2016/HASP16-16_slides.pdf

参考文献（2/2）

[8]”Towards TEEs with Large Secure Memory and Integrity Protection Against HW Attacks”, Pierre-Louis Aublin et al., https://systex22.github.io/papers/systex22-final15.pdf

[9]”32. Software Guard eXtensions (SGX) — The Linux Kernel documentation”, 2023/10/5閲覧, https://docs.kernel.org/arch/x86/sgx.html#encryption-engines

[10] Timestamp Cycle Counter (TSC), Intel, https://community.intel.com/t5/Intel-Software-Guard-Extensions/Timestamp-Cycle-Counter-TSC/m-p/1177414

[11] DuckDB-SGX2: The Good, The Bad and The Ugly within Confidential Analytical Query Processing, Ilaria Battiston et al., https://arxiv.org/pdf/2405.11988

[12] EnigMap: External-Memory Oblivious Map for Secure Enclaves, Afonso Tinoco et al., https://www.usenix.org/system/files/usenixsecurity23-tinoco.pdf