一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
CPSY2022-35,DC2022-94(2023-03)

ファットポインタとメモリプールを用いた柔軟かつ高効率な隔離実行環境のホスト・Enclave間データ授受手法

山本　希海† 　大森　侑† 　木村　啓二†

† 早稲田大学

E-mail: †{yamamoto_nozomi,oy}@kasahara.cs.waseda.ac.jp, ††keiji@waseda.jp

あらまし　これまで様々なOSの脆弱性が報告されており，秘匿性の高いデータを処理するプログラムの実行環境として，OSの信頼性は不確実なものとなっている．これに対して信頼実行環境の一つであるIntel SGXでは，Enclaveと呼ばれる隔離環境の中で重要なデータを扱うプログラムを実行することで，OSに対する悪性の攻撃などからデータを保護する．しかしながら，アプリケーションのホストとEnclaveの間で授受可能なデータ構造は現状では配列のような単純なものに限られており，ポインタを含むデータ構造を扱う場合は高コストなデータのシリアライズ処理を要する．本稿では，ファットポインタとメモリプールを用いることで，ポインタを含むデータ構造について柔軟かつ効率的なデータ授受を可能とする手法を提案する．本手法を用いてvectorとlistの授受を行ったところ，シリアライズを行った場合と比較してそれぞれ最大で18.52倍，19.64倍の速度向上が得られた．

キーワード　Intel SGX，Enclave，ファットポインタ，プールアロケータ，TOCTTOU攻撃

A Flexible and Efficient Data Transfer Method between Host and Enclave in Isolated Execution Environments Using Fat Pointers and Memory Pools

Nozomi YAMAMOTO†, Yu OMORI†, and Keiji KIMURA†

† Waseda University

E-mail: †{yamamoto_nozomi,oy}@kasahara.cs.waseda.ac.jp, ††keiji@waseda.jp

1. はじめに

OSはハードウェアの制御やプロセスの管理といったコンピュータにおける中核的な役割を果たす最も重要なソフトウェアの一つである．一方で，OSには数多くの脆弱性がこれまで存在し，それらに対する攻撃は多様で広範囲なものとなりつつある．例えばOSの脆弱性を狙ったランサムウェア攻撃の規模は近年拡大しており，その対象は企業や公共機関だけでなく個人にまで及ぶことが報告されている[1]．そのため秘匿性の高いデータを扱う場面において，OSは信頼できないという前提の下でプログラムを処理する実行環境が必要とされている．

このような背景から，信頼実行環境(Trusted Execution Environment: TEE)が注目されている．TEEの代表的な実装であるIntel SGX[2], [3]では，Enclaveと呼ばれるハードウェア的に保護された隔離環境をメモリ上に作成し，アプリケーション内の重要なデータを扱う処理をEnclaveに隔離して実行することでその安全性を保証する．

SGXを用いて秘密データの処理を行う際は，アプリケーションのホストからEnclaveへデータを確実に隔離することが重要である．しかし，現状のSGXがEnclaveとのデータ授受を安全に処理できるデータ構造は配列程度の単純なものに限られており，ポインタを含むデータ構造の授受を行うとデータの隔離が不十分な状態となる．そのため，複雑なデータ構造の授受においては事前にデータを配列などの形へシリアライズする必要があり，その処理に大きなコストを要する．

本稿では，ファットポインタとメモリプールを用いることでポインタを含むデータ構造について柔軟かつ効率的に授受を行う手法を提案する．また，本手法を用いてvectorとlistの授受を行った際の評価結果について報告する．

本稿の構成は以下の通りである．まず，2節でSGXプログラミングにおいて生じるセキュリティ上の問題点について説明し，その解決策として提案手法を3節で述べ，その性能を4節で報告する．5節では関連研究を紹介し，最後に6節でまとめる．

2. Intel SGX

Intel SGXはSkylake以降のIntel製CPUで利用可能なTEE

である．SGX では Enclave と呼ばれる隔離環境をメモリ上に作成し，アプリケーション内の重要なデータを扱う処理を Enclave に隔離して実行する．Enclave 内及び CPU・Enclave 間の通信は CPU 内に存在する MEE (Memory Encryption Engine) というユニットにより暗号化されており，さらにメモリ整合性により Enclave 内に存在するデータの整合性が保証されている．また，他プロセスだけではなく OS やハイパーバイザといった特権コードからの不正なアクセスを遮断するため，Enclave 内ではユーザ権限 (Ring3) の命令のみ実行が可能という制限がある．

2.1 SGX プログラミングの概要

SGX を用いた開発を行うプログラマのために，C/C++ による SGX SDK [4] が Intel により提供されている．前述のように Enclave 内ではシステムコールを呼び出すことができず，さらに Enclave 内で使用されるライブラリには潜在的なバグの原因となり得るコードが存在しないことが求められる．これらの理由から，SGX では Enclave 内における C/C++ 用の一般的なライブラリの使用が認められていない．そのため，標準ライブラリ機能の一部を移植した tlibc/tlibcxx という代替ライブラリが SGX SDK 内に用意されており，これらが C/C++ の共有ライブラリとして唯一利用を許可されている．

SGX SDK を用いたプログラミングでは通常の OS 上で実行するホスト処理と Enclave で実行する処理をそれぞれ関数単位で分離して実装し，Enclave 境界を越える処理は関数呼び出しという形で実行される．この時の，ホスト側から Enclave 側へ処理を移動するための関数呼び出しは ECALL，逆方向の関数呼び出しは OCALL と呼ばれる．例えば，ホストに存在する何らかのデータを Enclave 内で処理したい場合は，ECALL の引数にそのデータを指定して Enclave 内への値渡しを行う．

上記の ECALL/OCALL に基づく SGX のプログラム開発のために，SGX SDK には edger8r というツールが用意されている．ECALL と OCALL の関数定義を事前に EDL(Enclave Definition Language) というファイル形式で記述しておき，edger8r に入力することで，ECALL/OCALL に対応する SGX 命令である EENTER/EEXIT 命令を用いたグルーコードを出力する．これにより，プログラマは通常の関数と同じ構文で ECALL/OCALL を実装可能となる．

2.2 Enclave 境界を越えるポインタ変数の授受

SGX では Enclave 外の安全性を一切保証しないため，Enclave 内に存在するポインタ変数がホストのオブジェクトを参照する場合，参照先情報の書き換えといった攻撃を受ける危険性がある．また，ホストに存在するポインタ変数が Enclave 内を参照することは当然ながら不可能である．このような理由から，ECALL/OCALL の引数にポインタ変数を指定する場合は以下の 4 つのポインタ属性のいずれかを EDL ファイルの関数定義内に指定する必要がある．

- in
- out
- in, out
- user_check

各属性について，ECALL の引数ポインタに指定した際の動作を図 1 に示す．in 属性を指定した場合，ECALL の開始時にポインタ変数だけではなくその参照先データまで含めて Enclave 内にコピーされる．これにより，Enclave の中から外への危険な参照の発生を防いでいる．out 属性では，ECALL の開始時に Enclave 内にバッファが確保され，ECALL の終了時にバッファ内部のデータがホストへコピーされる．in, out 属性は，in 属性と out 属性の動作を兼ね備えた属性である．最後に，user_check 属性を用いた場合は通常の C/C++ のプログラム同様にポインタのみが授受される．

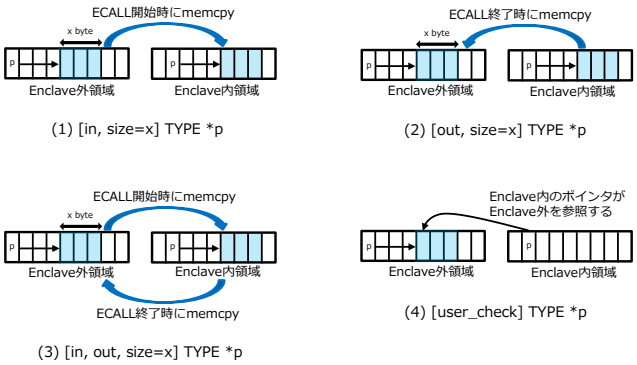

図 1　ECALL の引数ポインタにおける各属性の動作

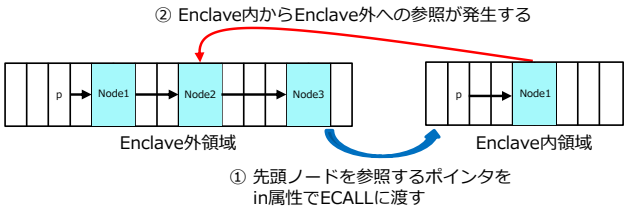

図 2　リスト構造を Enclave に in 属性で渡した際の動作

2.3 ポインタを含むデータ構造の授受で生じる問題

SGX SDK では，ポインタを含むデータ構造を扱う場合，2.2 節で述べた in 属性や out 属性では不十分である．例として，リスト構造の先頭ノードを参照しているポインタ変数を in 属性で ECALL の引数に指定した際の動作を図 2 に示す．この例ではポインタ変数 p が参照する Node1 は Enclave 内にコピーされるが，Node1 の中に含まれる Node2 へのポインタはディープコピーの対象とはならない．そのため Node2 以降は Enclave 内にコピーされず，データの隔離が不十分な状態となる．このような Enclave 境界を越えるポインタ参照については，Weichbrodt らの研究 [5] の中で TOCTTOU 攻撃への脆弱性に繋がるとしてその危険性が述べられている．

このような危険な参照の発生を防ぐためには，ECALL の呼び出し前にデータ構造を授受可能な形式にシリアライズする必要がある．代表的なシリアライズのライブラリとしては boost::serialization [6] や cereal [7] などが挙げられる．

例として，リスト構造の授受を行う際の整形処理を図 3 に示す．このようにホスト側でリストの走査を行い配列形式に整形してから ECALL に渡し，図 4 のように Enclave 内で再構築を行う必要があるため，大きなコストを要する．また，プログラマ定義のより複雑なデータ構造を扱う場合は，データの整形処

```
std::list<int> lst;
int *buf = (int *)malloc(sizeof(int) * lst.size());
int index = 0;

for(auto i = lst.begin(); i != lst.end(); i++){
  buf[index++] = *i;
}
```

図 3　std::list の整形処理

```
void ecall_rebuild_list(int *buf, int buf_size)
{
  int list_size = buf_size / sizeof(int);
  std::list<int> lst(buf, buf + list_size);

  return;
}
```

図 4　std::list の再構築処理

理はさらに複雑になり実行時間も大きくなる．

3. 提 案 手 法

3.1 想定する脅威

本稿は SGX SDK を用いたプログラミングの柔軟性向上という位置付けであるため，Tsai らの研究 [8] において述べられているような，SGX における典型的な脅威モデルを引き継ぐ．具体的には，プロセス内攻撃や悪性の OS による攻撃，コールドブート攻撃のような CPU 外のコンポーネントに対する攻撃などである．その中でも特に，実行中のアプリケーションについて攻撃者が Enclave 外領域に対する読み書きを自由に行うことができると仮定した際の TOCTTOU 攻撃を想定する．本稿の提案手法ではデータ構造内に含まれるポインタの参照先まで含めた領域を一つのバッファとしてコピーするため，この攻撃を受ける危険を自然な形で回避可能である．

3.2 想定するシナリオ

本手法は，SGX を使用するアプリケーションの開発者により用いられる．具体的には，アプリケーションのホストにリストなどのデータ構造に格納された何らかの秘密データが存在し，そのデータを用いた計算を安全に実行するため，ECALL の引数としてデータを Enclave に渡す場面を想定する．

3.3 メモリプールを用いたデータ授受

ポインタを含むデータ構造の安全な授受のためには，ポインタの参照先データまで含めた Enclave へのコピーが必要である．本手法では，メモリプールの概念を導入することでこれを実現する．なお，本稿においてはメモリプールという単語を，論理アドレス空間上で連続した巨大なメモリ領域として定義する．

図 5 に本手法を用いたリスト構造の授受の様子を示す．本手法ではポインタを含むデータ構造について，参照先オブジェクトまで含めて同一のプール上にあらかじめ配置することで，ポインタ参照がプール内で閉じている状態を作る．そして，プール先頭を指すポインタを in 属性もしくは in, out 属性で ECALL の引数に指定し，プール全体をバッファとして Enclave 内にコピーすることで，データ構造のシリアライズを回避する．これにより，ポインタを含む複雑なデータ構造について，その形式にかかわらず高速で画一的なデータ授受を可能とする．

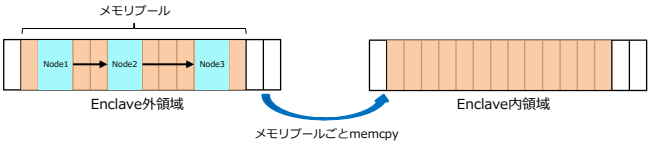

図 5　メモリプールを用いたデータ授受

```
typedef struct fp {
  uint64_t pool_id;
  uint64_t offset;
} FP;
```

図 6　ファットポインタの定義

3.4 メモリプール内で使用するポインタ表現

3.3 節で述べた手法により，データ構造内のポインタの参照先オブジェクトを Enclave 内にコピーすることが可能となった．しかしこの手法では，コピー後のプール内に存在するポインタ変数がコピー前のオブジェクトを参照してしまうという問題が生じる．そのため，プールのコピー後も正しい参照を保持することができるポインタ表現を導入する必要がある．

本稿では，ファットポインタと呼ばれる構造体形式のポインタ表現を導入する．本稿で用いるファットポインタの定義を図 6 に示す．pool_id メンバは参照先オブジェクトが属するプールの ID，offset メンバは参照先オブジェクトのプール先頭からのオフセットをそれぞれ保持する．

ファットポインタは構造体のため，通常のポインタとは異なり間接演算子などに対応していない．そのため，ファットポインタの参照先へアクセスするためには，以下の手順により絶対アドレス形式へのポインタ変換を行う必要がある．

（1） pool_id から参照先オブジェクトが属するプールの先頭アドレスを取得する．

（2） プールの先頭アドレスに offset を加算する．

pool_id からプールの先頭アドレスへの変換は，変換テーブルへのアクセスにより行われる．本手法では，アプリケーションのホストと Enclave それぞれにテーブルを用意し，ホストではプールの作成時に，Enclave ではホストからプールがコピーされた時にその先頭アドレスを登録することでテーブルを作成した．プールのコピーの前後でプール内オブジェクトの絶対アドレスは変化するが，プール先頭からの距離は変わらない．そのため，プール内のポインタ表現としてファットポインタを用いることで，3.3 節の手法において生じる問題の解決が可能となる．

3.5 提案手法の実装

本手法の実現のためには，ファットポインタの利用を想定したプールアロケータが必要である．具体的には，アロケータは以下の 2 つの要件を満たしている必要がある．

- プールのメモリ管理情報は全てそのプールの中に配置される．

表 1　評 価 環 境

OS	Ubuntu 20.04 LTS
CPU	Intel(R) Xeon(R) E-2136
動作周波数	3.30GHz
L1d Cache	32KiB/core
L1i Cache	32KiB/core
L2 Cache	256KiB/core
L3 Cache	12MiB/processor
EPC	96MiB

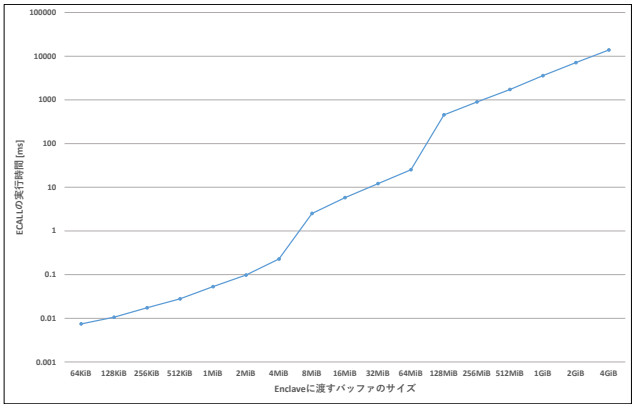

図 7　in 属性のポインタ授受に要する時間

- メモリ管理情報内で用いられるポインタは全てファットポインタ形式である．

ファットポインタを用いたプールアロケータとしては，Intel より不揮発性メモリアプリケーション開発用に提供されている libpmemobj [9] のものが存在するが，libpmemobj は上記の 1 つ目の要件を満たさないためプールのコピーに対応していない．そこで文献 [10] で用いられている記憶割当てアルゴリズムを使用し，上記の要件を満たしたアロケータのプロトタイプを実装した．

4. 評　　価

4.1 評 価 環 境

本稿の評価で使用した評価環境を表 1 に示す．表 1 内の EPC(Enclave Page Cache) は Enclave からメタデータなどを除いた，アプリケーションが利用可能な領域のことを指す．形式的には EPC サイズが Enclave 内で利用可能な領域の上限サイズとされているが，Linux 環境においては OS のページング機能により，実際には EPC サイズ以上の領域を扱うことが可能である．

また，本評価では計測コードを繰り返し実行してキャッシュが十分に温まった状態で計測を行った．

4.2 ECALL のバッファコピー性能の評価結果

提案手法の評価に先立ち，SGX におけるデータ授受の性質を調べるため，ECALL で生じるバッファコピーの性能を評価した．関数内で何もせずにすぐさま return する ECALL に対し，in 属性及び in, out 属性のポインタ変数を渡した際の実行に要した時間を図 7，8 にそれぞれ示す．2.2 節で述べたように，in 属性では 1 回，in, out 属性では 2 回のバッファコピーが発生する．

図 7，8 では共に，バッファサイズが 64MiB から 128MiB

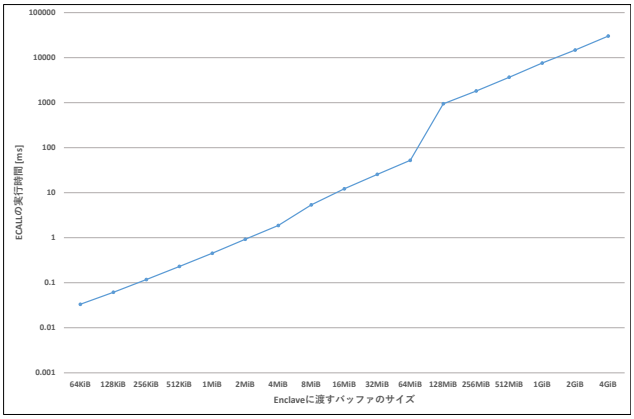

図 8　in, out 属性のポインタ授受に要する時間

になった際に実行時間が約 18 倍と大幅に増加する．これは，Enclave 内で使用したヒープ領域が EPC サイズを超えたことにより，ページング処理が発生したためである．

また，図 7 ではバッファサイズが 4MiB の時と 8MiB の時で実行時間の隔たりが見られるが，これは L3 キャッシュによる影響であると考えられる．評価マシンの L3 キャッシュは 12MiB であり，バッファサイズが 8MiB の時はバッファのコピー元領域とコピー先領域で計 16MiB のアクセスが発生するため，キャッシュミスにより実行時間が増加した．一方で，図 8 ではキャッシュによる影響は僅かにしか確認できない．これは，ECALL 終了時に実行される Enclave からホストへのバッファコピー処理が，評価で使用した SDK のバージョンでは非効率的な実装になっており，実行時間の小さい 4MiB 以下の場合においてそのオーバーヘッドが顕著に現れたためである．

4.3 提案手法の評価に用いるデータ構造

ポインタを含む一般的なデータ構造として int 型の可変長配列及びリストを選択し，これらの授受についてデータの整形処理を行う場合と提案手法を用いる場合の比較による評価を行った．

データの整形処理を行う場合の計測には，C++ の標準ライブラリ及び SGX SDK の tlibcxx で実装されている std::vector と std::list を使用した．一方で，提案手法の評価では授受対象のデータ構造が以下の条件を満たしている必要があるため，標準ライブラリで実装されているものを使用することができない．

- データ構造内のポインタは全てファットポインタ形式である．
- データ構造への要素の追加・削除はプールアロケータにより行われる．

そのため上記の条件を満たした可変長配列（以下 tee::vector とする）とリスト（以下 tee::list とする）を実装し，提案手法の計測に使用した．

4.4 提案手法の評価内容

4.4.1 in 属性のデータ授受の評価

ホストに存在するデータ構造を in 属性のバッファとして Enclave に渡し，Enclave 内で元のデータ構造の形式に直すまでに要した時間と Enclave 内のヒープ使用量を計測した．std::vector についてはメンバ関数 data により得られる内部配列の先頭ポイ

ンタを ECALL の引数として用い，Enclave 内でコンストラクタを呼び出すことで再構築を行った．std::list は図 3 の手順で配列へ変換して授受を行い，std::vector と同様にコンストラクタによる再構築を行った．これに対し，提案手法ではデータ構造が収まる最小サイズのプールを ECALL で渡し，Enclave 内でプールからデータ構造を取り出すまでの操作について計測した．

4.4.2 in, out 属性のデータ授受の評価

ホストに存在するデータ構造を Enclave に渡し，Enclave 内で何らかの処理を行ったと仮定して，そのデータ構造をホストへ戻す操作における実行時間及び Enclave ヒープ使用量を計測した．提案手法ではプール上に評価対象のデータ構造だけでなくアロケータのメモリ管理情報まで乗せているため，ECALL に渡すバッファの属性を in, out にすることでこの動作を実現できる．一方，std::vector と std::list についてはホストで整形して得られたバッファを in, out 属性で渡して Enclave 内で再構築を行った後に，Enclave 内で再度そのデータ構造を配列に変換して ECALL の引数バッファにコピーし，ECALL 終了後にホストに書き戻されたバッファからデータ構造を再構築するまでの操作を測定対象とした．

4.5 vector 授受の評価結果

in 属性の vector 授受を行った際の実行時間を表 2 に示す．この表から，vector の要素数が少ない場合に最大で 18.52 倍の速度向上が得られていることが分かる．これは，提案手法で行っているのは単純なバッファのコピーであるため，4.2 節で述べたキャッシュの恩恵を大きく受けることができたためである．一方で要素数が 10,000,000 になると速度向上率が 4.70 倍となりそれまでと比べて低下しているが，これは提案手法において Enclave ヒープの使用量が L3 キャッシュサイズを超えたためである．そして，要素数が 100,000,000 になると Enclave ヒープ使用量が EPC サイズを超えるため，提案手法を用いる場合とデータ整形を行う場合の両方で実行時間が増加し，それぞれ 10,000,000 の時に比べて 92 倍，65 倍になる．

次に in, out 属性の授受を行った際の実行時間を表 3 に示す．提案手法の速度向上率は最大 3.99 倍となった．4.2 節で述べたように in, out 属性ではキャッシュによる影響が僅かであるため，in 属性の結果と比較して速度向上率は小さい．

また，vector 授受に要した Encalve ヒープ量を表 4 に示す．この表は，in 属性と in, out 属性で共通の結果である．この表より，提案手法を用いた場合において Enclave ヒープ使用量を半分に抑えられていることが分かる．これは，データ整形を行う場合は vector の再構築を行うために Enclave ヒープを確保する必要があるためである．

4.6 list 授受の評価結果

in 属性の list 授受を行った際の実行時間を表 5 に示す．この結果においても L3 キャッシュの影響により，要素数が少ない時に最大で 19.64 倍の速度向上が得られた．また，list の授受では要素数が 10,000,000 になると Enclave ヒープ量が EPC サイズを超え，実行時間の大部分がページング処理に支配される結果となった．この際，後述の理由により提案手法では Enclave ヒープ使用量が大きくなることで実行時間が増加し，速度向上率は 2 倍程度となった．

表 2　in 属性の vector 授受に要した実行時間

要素数	実行時間 [ms]		速度向上率
	std::vector	tee::vector	
10,000	0.042	0.007	6.00
100,000	0.379	0.030	12.63
1,000,000	4.575	0.247	18.52
10,000,000	68.622	14.590	4.70
100,000,000	4430.742	1338.863	3.31

表 3　in, out 属性の vector 授受に要した実行時間

要素数	実行時間 [ms]		速度向上率
	std::vector	tee::vector	
10,000	0.061	0.021	2.90
100,000	0.568	0.172	3.30
1,000,000	6.835	1.711	3.99
10,000,000	119.629	30.938	3.87
100,000,000	8950.871	2823.285	3.17

表 4　vector 授受に要した Enclave ヒープ量

要素数	Enclave ヒープ ピーク使用量 [KiB]		ヒープ使用率
	std::vector	tee::vector	
10,000	80	40	0.50
100,000	784	392	0.50
1,000,000	7816	3908	0.50
10,000,000	78128	39064	0.50
100,000,000	781252	390628	0.50

次に，in, out 属性の授受に要した実行時間を表 6 に示す．提案手法の速度向上率は最大 5.34 倍となった．L3 キャッシュの影響が小さいため，in 属性の時と比較して速度向上率は小さい．一方で，要素数が大きくなるほど速度向上率が小さくなるという点で同様の傾向が見られた．

また，list 授受に関する Enclave ヒープ使用量を表 7 に示す．この結果も vector についての結果と同様に，in 属性と in, out 属性で共通のものである．この表から list の授受では，提案手法において Enclave ヒープの使用量が最大で 1.24 倍に増加していることが分かる．この原因として，ファットポインタが 16byte であることによるメモリ使用量の増加が挙げられる．tee::list では，自身の前後の要素を参照するために各ノードが 2 つのファットポインタを有する．これに加えて，プールアロケータの実装に用いた記憶割当てアルゴリズムの非効率性も原因として挙げられる．このアルゴリズムではプールから領域を確保する際に 16byte のヘッダを付け加え，さらに最終的な要求サイズが 16 の倍数になるようにサイズを切り上げる必要がある．結果，tee::list ではノード 1 個あたり 64byte の領域を必要とする．一方，tlibcxx で実装されている std::list ではノード 1 個に要する領域は 48byte で済むため，提案手法を用いた場合と比較して Enclave ヒープ使用量が少なくなる．

本評価においては std::list の整形・再構築に要するコストの削減効果が Enclave ヒープ使用量の増加によるオーバーヘッド

表 5　in 属性の list 授受に要した実行時間

要素数	実行時間 [ms]		速度向上率
	std::vector	tee::vector	
10,000	0.864	0.044	19.64
100,000	8.491	1.153	7.36
1,000,000	90.953	23.862	3.81
10,000,000	4458.147	2070.362	2.15
100,000,000	44449.809	21441.563	2.07

表 6　in, out 属性の list 授受に要した実行時間

要素数	実行時間 [ms]		速度向上率
	std::vector	tee::vector	
10,000	1.464	0.274	5.34
100,000	14.649	3.421	4.28
1,000,000	156.074	49.828	3.13
10,000,000	6636.977	4396.399	1.51
100,000,000	67718.280	43547.931	1.56

表 7　list 授受に要した Enclave ヒープ量

要素数	Enclave ヒープ ピーク使用量 [KiB]		ヒープ使用率
	std::vector	tee::vector	
10,000	508	628	1.24
100,000	5080	6252	1.23
1,000,000	50784	62504	1.23
10,000,000	507816	625004	1.23
100,000,000	5078128	6250004	1.23

を上回ったため，提案手法において速度向上を得ることができた．このオーバーヘッドは，アロケータの性能向上やより効率的なファットポインタ表現 [11], [12] の導入により改善が可能である．また，今回は int 型を評価に用いたが，より実用的でサイズの大きい型を使う場合は，ファットポインタによる影響が相対的に小さくなると考えられる．

5. 関 連 研 究

SGX におけるデータ授受の高速化手法として HotCalls [13] が提案されている．この手法では Enclave 内外の処理を別のスレッドに分けて実行し，さらにメモリの一部をデータ授受のための通信チャネルとして使用している．これにより，ECALL/OCALL で生じるコンテキストスイッチを削減して高速化を実現しているが，ポインタを含むデータ構造の扱いやシリアライズに要するコストについては議論されていない．

ポインタを含むデータ構造の安全な授受に関する手法として Cadote [14] が提案されている．この手法では SGX プログラミングのインタフェースとして Rust を指定し，Rust コンパイラの静的解析機能を活用して授受対象のデータ構造に含まれるポインタに対するディープコピーを実現している．また類似の研究として，入力言語として Go を使用することを想定し，Go コンパイラにディープコピーセマンティクスを実装した GOTEE [15] という手法も存在する．しかし，これらの手法はいずれも入力言語のコンパイラ機能に依存しており，さらにデータ授受の高速化については十分な言及がないという点で本稿とは異なる．

6. ま と め

本稿では，Intel SGX を用いるアプリケーションのホスト・Enclave 間のデータ授受で生じていた，速度と安全性の問題の両面において有効なデータ授受手法を提案した．ポインタを含むデータ構造について，メモリプールのコピーという形でデータ授受を行うことで高速化を実現し，さらにプール内のポインタ表現としてファットポインタを用いることで危険な参照の発生を防いだ．

本手法を用いて C++ の vector と list についてデータ授受を行ったところ，それぞれ最大で 18.52 倍，19.64 倍の速度向上が得られた．また，list の授受では実装したアロケータの性能とファットポインタの影響により Enclave 内のヒープ使用量が最大で 1.24 倍に増加したものの，vector の授受においては 0.50 倍に削減できることを確認した．

文　献

[1] M.A. Vander-Pallen, P. Addai, S. Isteefanos, and T.K. Mohd, “Survey on types of cyber attacks on operating system vulnerabilities since 2018 onwards,” 2022 IEEE World AI IoT Congress (AIIoT)IEEE, pp.01–07 2022.

[2] F. McKeen, I. Alexandrovich, A. Berenzon, C.V. Rozas, H. Shafi, V. Shanbhogue, and U.R. Savagaonkar, “Innovative instructions and software model for isolated execution.,” vol.10, no.1, Hasp@ isca, 2013.

[3] V. Costan and S. Devadas, “Intel sgx explained,” , Cryptology ePrint Archive, 2016.

[4] Intel, “linux-sgx”. https://github.com/intel/linux-sgx/

[5] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, “Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves,” Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21Springer, pp.440–457 2016.

[6] Boost.org, “Boost C++ Libraries”. https://www.boost.org/

[7] iLab at University ofSouthern California, “cereal - A C++11 library for serialization”. https://uscilab.github.io/cereal/

[8] C.-C. Tsai, D.E. Porter, and M. Vij, “Graphene-sgx: A practical library os for unmodified applications on sgx.,” USENIX Annual Technical Conference, pp.645–658, 2017.

[9] Intel, “libpmemobj”. https://pmem.io/pmdk/libpmemobj/

[10] B.W. カーニハン (著)，D.M. リッチー (著)，石田晴久 (訳)，プログラミング言語 C 第 2 版 ANSI 規格準拠，共立出版株式会社，1989.

[11] C. Ye, Y. Xu, X. Shen, X. Liao, H. Jin, and Y. Solihin, “Supporting legacy libraries on non-volatile memory: a user-transparent approach,” 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)IEEE, pp.443–455 2021.

[12] G. Chen, L. Zhang, R. Budhiraja, X. Shen, and Y. Wu, “Efficient support of position independence on non-volatile memory,” Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, pp.191–203, 2017.

[13] O. Weisse, V. Bertacco, and T. Austin, “Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves,” vol.45, no.2, ACM SIGARCH Computer Architecture News, 2017.

[14] F. Dreissig, J. Röckl, and T. Müller, “Compiler-aided development of trusted enclaves with rust,” Proceedings of the 17th International Conference on Availability, Reliability and Security, pp.1–10, 2022.

[15] A. Ghosn, J. Larus, and E. Bugnion, “Secured routines: Language-based construction of trusted execution environments,” Proceedings of the 2019 Usenix Annual Technical Conference, no.CONFUSENIX ASSOC, pp.571–585 2019.