Arm TrustZoneのワールド間における POSIX APIを用いた安全な協調実行

佐藤 太陽[1]　光来 健一[1]

概要：近年，クラウドのアプリケーションをユーザの近くで実行するエッジコンピューティングが普及してきている．信頼できないエッジデバイスにおいても Arm TrustZone を用いることにより，クラウドアプリケーションをセキュアワールドで TA として安全に実行することができる．しかし，セキュアワールドは高い権限を持つため，アプリケーションに脆弱性があるとシステム全体に影響が及ぶ恐れがある．そのため，アプリケーションの大部分はノーマルワールドで CA として実行するのが望ましいが，CA と TA を連携させるには専用 API を用いる必要があり，柔軟な協調は容易ではない．本稿では，アプリケーションを TrustZone の 2 つのワールドに分割し，ワールド間で POSIX API を用いて協調実行を行うシステム TZmediator を提案する．TZmediator は保護する必要がある処理のみをセキュアワールドで実行し，それ以外の処理はノーマルワールドで実行する．ワールド間にまたがって通信を行えるようにするために，ノーマルワールド内に TA に対応するシャドウプロセスを作成し，TA による POSIX API の呼び出しを代理で実行させる．OP-TEE を用いて TZmediator を実装し，CA と TA 間で POSIX API を用いてデータを送受信する際の通信性能を調べた．

1. はじめに

近年，クラウドのアプリケーションをユーザの近くで実行することによりサービスの品質を向上させるエッジコンピューティングが普及してきている．エッジデバイスにおいては OS すら信頼できない場合があるが，CPU が提供する Trusted Execution Environment (TEE) を用いることでクラウドアプリケーションを安全に実行することができる．TEE はメモリを論理的に隔離するが，メモリの暗号化も行う場合には物理的な盗聴も困難となる．主要な CPU で実装されている TEE アーキテクチャには Intel SGX [1] や AMD SEV [2]，Arm TrustZone [3]，RISC-V Keystone [4] などがある．

エッジデバイス向けの TEE である Arm TrustZone は 2 つのワールドを提供し，セキュアワールドで Trusted Application (TA) が動作し，ノーマルワールドで Client Application (CA) が動作する．クラウドアプリケーションをセキュアワールドに隔離して実行することが考えられるが，セキュアワールドは高い権限を持っているため，クラウドアプリケーションに脆弱性があるとシステム全体に影響が及ぶ恐れがある．そのため，TA として実行する部分をできるだけ小さくして，CA と連携しながらクラウドアプリケーションを実行することが望ましい．しかし，CA と TA を連携させるには専用の API を用いる必要があり，CA と TA の柔軟な協調は容易ではない．

そこで本稿では，クラウドアプリケーションを TrustZone の 2 つのワールドに分割し，ワールド間で POSIX API を用いて協調実行を行うシステム TZmediator を提案する．TZmediator は保護する必要がある処理のみをセキュアワールドで TA として実行し，それ以外の処理はノーマルワールドで CA として実行する．ワールド間にまたがって POSIX API を用いた通信を行えるようにするために，ノーマルワールド内に TA に対応するシャドウプロセスを作成する．TA が POSIX API を実行する際にはシャドウプロセスを呼び出して標準 C ライブラリの API を代理実行させる．TA をさらに安全に実行する必要がある場合には WebAssembly を用い，WASI を拡張して CA との通信を行う．

TZmediator を OP-TEE に実装し，TA と CA にそれぞれ提供される専用ライブラリを開発した．TA 用の専用ライブラリは CA と通信を行うために必要な POSIX API を提供し，遠隔手続き呼び出し (RPC) を用いて tee-supplicant のプラグインとして実装されたシャドウプロセスを呼び出す．現在のところ，名前付きパイプ，UNIX ドメインソケット，ネットワークソケットに対応している．CA 用の

[1] 九州工業大学
Kyushu Institute of Technology

© 1959 Information Processing Society of Japan

専用ライブラリはサブスレッドを作って TA のコマンドを実行し続けることにより，CA と TA の並列実行を可能にする．

TZmediator を用いて CA と TA が双方向に通信できることを確認する実験を行った．その結果，名前付きパイプ，UNIX ドメインソケット，TCP ソケットを用いて CA と TA それぞれからデータを送信し，通信相手が正常にデータを受信できることが確認できた．また，通信の往復レイテンシとスループットを測定し，専用 API を用いて共有メモリ経由で通信を行った場合との比較を行った．TZmediator は専用 API を用いる場合よりも往復レイテンシが 1.6〜2.4 倍に増加し，スループットが 75〜78%低下することが分かった．

以下, 2 章で Arm TrustZone を用いてクラウドアプリケーションを実行する上での問題点を述べる．3 章で POSIX API を用いてワールド間でクラウドアプリケーションの協調実行を可能にするシステム TZmediator を提案する．4 章で TZmediator の実装について述べ, 5 章で TZmediator を用いて行った実験について述べる．6 章で関連研究について述べ，7 章で本稿をまとめる．

2. TrustZone を用いた安全な実行

クラウドのアプリケーションをユーザの近くで実行するエッジコンピューティングは，クラウドで実行する場合と比べて低遅延・高速・リアルタイム処理が可能となる．また，クラウドとの通信を減らせることから，ネットワーク負荷を軽減することもできる．しかし，IoT 機器やスマートフォンなどのエッジデバイスはクラウドの管理外であるため，搭載している OS を含めて必ずしもセキュリティが強固であるとは限らない．加えて，通常のエッジデバイスはリソースが限られており，強力なセキュリティ機能を実装することが難しい．これらのことから，エッジデバイスはリバースエンジニアリングや機微情報の窃取，処理の改竄などの攻撃を受けやすくなる可能性がある．

このようなエッジデバイスにおいてクラウドアプリケーションを安全に実行するために，CPU によって提供されている Trusted Execution Environment (TEE) を用いることが考えられる．TEE はメモリを論理的に隔離し，その上でプログラムの実行を行うことを可能にする．主要な CPU で実装されている TEE アーキテクチャには Intel SGX [1] や AMD SEV [2]，Arm TrustZone [3]，RISC-V Keystone [4] などがある．TEE を用いることで，OS やハイパーバイザに脆弱性があったとしてもクラウドアプリケーションを安全に実行することができる．さらに，隔離したメモリを暗号化する TEE では物理的なメモリの盗聴を防ぐこともできる．

エッジデバイス向けの Arm TrustZone はワールドと呼ばれる実行環境を 2 つ提供する．セキュアワールドが TEE であり，Trusted Application (TA) と呼ばれるアプリケーションを安全に実行することができる．TA を実行するための Trusted OS もセキュアワールド内で動作する．Trusted OS としては，OP-TEE [5] や Open-TEE [6]，Trusty [7] などがある．一方，ノーマルワールドは Rich Execution Environment (REE) と呼ばれる通常の実行環境であり，通常のアプリケーションおよび TA を利用するための Client Application (CA) が動作する．ノーマルワールド内ではこれらのアプリケーションを実行するための Linux 等の Rich OS が動作する．ワールド間の切り替えはモニタによって行われ，システムリソースは厳密に分離される．そのため，ノーマルワールドはセキュアワールド用に確保されたリソースにアクセスすることはできない．

しかし，TrustZone のセキュアワールドでクラウドアプリケーションを実行すると 2 つの問題が生じる．1 つはセキュアワールドがノーマルワールドより高い権限を持つことであり，もう 1 つは複数の TA が一つの実行環境で動作することである．エッジデバイスで実行するクラウドアプリケーションは従来，TA として実行していたアプリケーションよりも高機能になることが多いと考えられるため，脆弱性がある可能性が高くなる．ノーマルワールドに侵入した攻撃者にその脆弱性を利用した攻撃を行われると，セキュアワールドの権限を奪われる恐れがある．実際に，TA や Trusted OS には様々な脆弱性があることが報告されている [8]．Trusted OS が攻撃を受けるとノーマルワールドのメモリを盗聴できるなど，システム全体に影響が及ぶ可能性がある．一般的に，クラウドアプリケーションは高い権限を必要としているわけではなく，隔離を必要としているだけである．また，セキュアワールド内で他の TA を攻撃することにより，TA の持つ機密情報を盗むことも可能になる．

クラウドアプリケーションをより安全に実行できるようにするには，セキュアワールド内で TA として実行する部分をできるだけ小さくすることが望ましい．その場合，機密情報や機密性の高いアルゴリズムのみを TA に含め，それ以外はノーマルワールドで CA として実行することになる．TrustZone では，CA と TA を連携させて実行するために専用 API を用いる必要がある．セキュアワールドで実行される TA は，GlobalPlatform [9] が定義する TEE Internal Core API [10] に従わなければならない．一方，ノーマルワールドで実行される CA は，TA のクライアントとして動作するために TEE Client API [11] に従わなければならない．これらの API は標準化されているものの，通常の OS で使われる POSIX API とは大きく異なる．例えば，TA は CA からのコマンド呼び出しによって実行され，実行が終了するまで CA は待たされる．そのため，CA と TA との柔軟な協調ができず，クラウドアプリケーションの開発が難しい．

© 1959 Information Processing Society of Japan

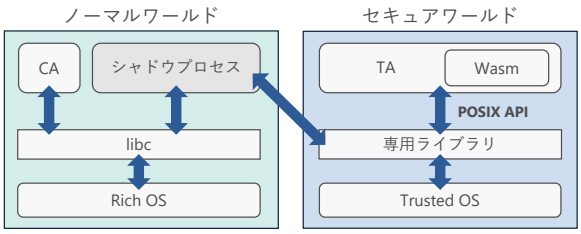

図 1 TZmediator のシステム構成

3. TZmediator

本稿では，クラウドアプリケーションを 2 つのワールドに分割し，ワールド間で POSIX API を用いて協調実行を行うシステム TZmediator を提案する．TZmediator は保護する必要がある処理のみをセキュアワールドで TA として実行し，それ以外の処理はノーマルワールドで CA として実行する．そして，CA と TA を並列に実行し，通信によって協調する．

3.1 脅威モデル

TrustZone の機能を提供するハードウェアに脆弱性はなく，TrustZone による保護は破られないことを仮定する．TrustZone ではセキュアワールドのメモリは暗号化されないため，メモリに対する物理的な攻撃は対象としない．セキュアワールドでは，Trusted OS などの高い権限で動作するソフトウェアには脆弱性がないものとする．また，クラウドアプリケーションのセキュアワールドで動作する部分は信頼できるものとし，ノーマルワールドから攻撃できる脆弱性は持たないものとする．一方，ノーマルワールドでは，Rich OS を含めてすべてのソフトウェアは信頼できないものとする．本稿では，外部やノーマルワールド内の攻撃者がクラウドアプリケーション内の機密情報を盗む攻撃を想定する．クラウドアプリケーションに対するそれ以外の攻撃は対象としない．

3.2 POSIX API を用いた協調実行

TZmediator はワールド間にまたがって実行される CA と TA が POSIX API を用いて通信を行えるようにするために，TA に対応するシャドウプロセスをノーマルワールド内に作成する．TZmediator のシステム構成を**図 1** に示す．シャドウプロセスはノーマルワールドで提供される標準ライブラリを用い，Rich OS に対してシステムコールを発行することができる．これらのセマンティクスは通信相手の CA と同じであるため，POSIX に対する高い互換性を実現することができる．CA と TA 間の通信に用いられる POSIX API として，名前付きパイプや UNIX ドメインソケット，ネットワークソケットなどのための API が挙げられる．

TA による POSIX API の実行の流れを**図 2** に示す．セキュアワールドで動作する TA は TZmediator が提供する専用ライブラリ経由で POSIX API を利用する．このライブラリは TA に提供される TEE Internal Core API を用いて遠隔手続き呼び出し (RPC) を実行し，ノーマルワールドのシャドウプロセスを呼び出す．シャドウプロセスは CA に提供される TEE Client API を用いて RPC のリクエストを受け取り，標準 C ライブラリを用いて指定された POSIX API を代理実行する．その実行結果は RPC 経由でセキュアワールドの TA に返送される．このように，専用 API を用いたワールド間通信は専用ライブラリとシャドウプロセスによってクラウドアプリケーションから隠蔽される．

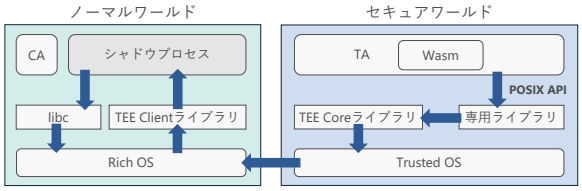

図 2 TA による POSIX API 実行

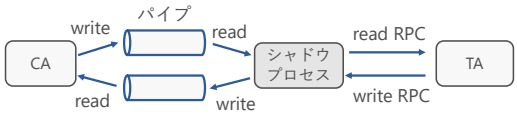

図 3 パイプを用いた通信の例

例として，TA が名前付きパイプを用いて CA と通信を行う場合の処理の流れを**図 3** に示す．TA が名前付きパイプを作成する場合，作成要求をシャドウプロセスに送信し，シャドウプロセスがノーマルワールドに名前付きパイプを作成する．TA がパイプにデータを書き込む場合，書き込み要求をシャドウプロセスに送信し，シャドウプロセスがノーマルワールドに作成されたパイプにデータを書き込む．一方，TA がパイプからデータを読み込む場合，読み込み要求をシャドウプロセスに送信し，シャドウプロセスがノーマルワールドのパイプからデータを読み込む．そして，そのデータをセキュアワールドの TA に返送する．

CA による POSIX API の実行の様子を**図 4** に示す．ノーマルワールドで動作する CA は標準 C ライブラリによって提供される POSIX API を利用して，TA の代理として作成されるシャドウプロセスと通信を行う．さらに，シャドウプロセスが TA と通信を行うことによって，CA と TA 間の通信を実現する．例えば，CA が名前付きパイプにデータを書き込んだ時，シャドウプロセスがパイプからそのデータを読み込み，TA にデータを送信する．CA がパイプからデータを読み込む際には，TA がシャドウプロセス経由でパイプに書き込んだデータを読み込む．

TZmediator はセキュアワールドで実行される TA をできるだけ小さくするが，さらに安全性を高める必要があ

© 1959 Information Processing Society of Japan

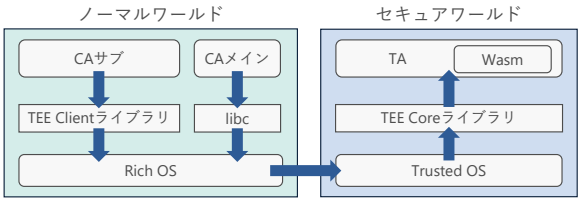

図 4 CA による POSIX API 実行

る場合には WebAssembly (Wasm) を用いて TA を実行する [12]．Wasm は様々なプログラミング言語で開発されたアプリケーションをサンドボックス内で安全に実行することができるため，クラウドアプリケーションが攻撃を受けたとしても Trusted OS や他の TA を攻撃するのは難しい．Wasm アプリケーションは WebAssembly System Interface (WASI) と呼ばれる POSIX に似た API を用いることができる．TZmediator は WASI を拡張して POSIX API との互換性を持たせ，Wasm ランタイムが提供する WASI ライブラリにおいて RPC を用いてシャドウプロセスを呼び出す．これにより，セキュアワールドの Wasm アプリケーションがノーマルワールドの CA と通信を行うことができる．

4. 実装

TZmediator を OP-TEE 4.0.0 [13] に実装した．OP-TEE は TA をセキュアワールドで動作させるための Trusted OS である OP-TEE OS や，GlobalPlatform API に準拠したライブラリを提供する．ノーマルワールドで動作する Rich OS には Linux を用い，Linux 上で TA からのリクエストを処理するデーモンである tee-supplicant を動作させる．

4.1 シャドウプロセス

シャドウプロセスを tee-supplicant のプラグインとして実装した．プラグインは TEE Client API を用いて共有ライブラリとして作成され，tee-supplicant の起動時にロードされる．TA からプラグインへのすべての RPC リクエストは tee-supplicant の共通の RPC ハンドラによって処理される．この RPC はセキュアワールドとプラグイン間で統一されたインタフェースとなっている．プラグインには一意の UUID を割り当て，TA は UUID を用いてシャドウプロセスのプラグインを指定する．tee-supplicant は RPC リクエストを受け取ると，UUID に対応するプラグインの関数を実行する．

TA は RPC を用いてシャドウプロセスを呼び出す際に，TEE Internal Core API を用いて POSIX API の種類と引数を渡す．POSIX API の種類はプラグインによって用いられるコマンドとして指定し，引数は TA とプラグイン間でデータをやり取りするために用いられる RPC 用バッファの中に格納する．このバッファには TA と CA の間の通信に用いられる様々な POSIX API のすべての引数を含む構造体が格納される．例えば，ファイル記述子，ソケットアドレス構造体，読み書きするデータなどである．データのサイズは POSIX API の呼び出しごとに異なるため，構造体のフレキシブル配列メンバを用いる．この構造体には，POSIX API の通信に必要な引数が含まれており，シャドウプロセスでの POSIX API の実行結果もこの構造体に格納される．

シャドウプロセスが標準 C ライブラリの POSIX API を呼び出すと，Rich OS に対して一般ユーザの権限でシステムコールが発行される．これは tee-supplicant が一般ユーザの権限で動作しているためである．CA が名前付きパイプなどを作る際には tee-supplicant の権限で読み書きができるようにする必要がある．

4.2 TA に提供する専用ライブラリ

TA として動作することを意識せずにクラウドアプリケーションを開発できるようにするために，TA に提供する専用ライブラリにおいて必須のエントリポイント [14] を実装した．この専用ライブラリは TEE Internal Core API v1.3.1 に準拠している．まず，TA のインスタンスが作成される際に TA_CreateEntryPoint 関数が呼び出される．次に，CA が新しいセッションを開くために TA のインスタンスに接続しようとした際に TA_OpenSessionEntryPoint 関数が呼び出される．その後，CA が TA の開始コマンドを実行すると TA_InvokeCommandEntryPoint 関数が呼び出され，クラウドアプリケーション内で定義された main 関数が呼び出される．以降はクラウドアプリケーションが動作し続け，その実行が終了した時にコマンドの実行も完了する．CA の実行が終了する際にセッションを閉じると TA_CloseSessionEntryPoint 関数が呼び出される．最後に，TA のインスタンスが終了する直前に TA_DestroyEntryPoint 関数が呼び出される．

この専用ライブラリは TA が CA と通信を行うために必要な POSIX API を提供する．現在のところ，名前付きパイプ，UNIX ドメインソケット，ネットワークソケットに対応している．

4.2.1 名前付きパイプ

TA が名前付きパイプを用いて CA との通信できるようにするために実装した POSIX API の一覧を**表 1** に示す．TA が名前付きパイプを作成する際には専用ライブラリの mkfifo 関数が呼び出される．この関数は引数で指定されたパス名を格納できるサイズの RPC 用バッファを確保し，パス名とモードを格納する．そして，RPC を実行して tee-supplicant のプラグインとして実装されたシャドウプロセスを呼び出す．シャドウプロセスは渡されたパス名とモードを用いて標準 C ライブラリの mkfifo 関数を呼び出し，ノーマルワールドに名前付きパイプを作成する．そ

© 1959 Information Processing Society of Japan

表 1　名前付きパイプのための POSIX API

関数	動作
mkfifo	名前付きパイプを作成
open	名前付きパイプを開く
write	名前付きパイプにデータを書き込む
read	名前付きパイプからデータを読み込む
close	名前付きパイプを閉じる
unlink	名前付きパイプを削除

の後，mkfifo 関数の実行に成功したか失敗したかの情報を RPC 用バッファに格納し，専用ライブラリ経由で TA に返り値として返す．TA が名前付きパイプを削除する際には unlink 関数が呼び出される．この関数は指定されたパス名をシャドウプロセスに渡し，シャドウプロセスがノーマルワールドの名前付きパイプを削除する．

TA は名前付きパイプのパス名を指定して専用ライブラリの open 関数を呼び出すことにより，名前付きパイプを開く．この関数は mkfifo 関数と同様に，引数で指定されたパス名とフラグを RPC 用バッファに格納してシャドウプロセスを呼び出す．シャドウプロセスは渡されたパス名に対して標準 C ライブラリの open 関数を実行してノーマルワールドに作成された名前付きパイプを開き，その返り値のファイル記述子をそのまま TA に返す．TA はそのファイル記述子を用いて専用ライブラリの write 関数や read 関数を呼び出し，名前付きパイプの読み書きを行う．write 関数では渡されたデータが格納できるサイズの RPC 用バッファを確保し，データをそのバッファにコピーする．read 関数では読み込むデータが格納できるサイズの RPC 用バッファを確保する．そして，シャドウプロセスを呼び出して標準 C ライブラリの write 関数や read 関数を実行する．read 関数で名前付きパイプから読み込んだデータは RPC 用バッファに格納され，専用ライブラリが TA から渡されたバッファにコピーすることにより TA に返される．TA が開いた名前付きパイプを閉じる際には close 関数が呼び出され，指定されたファイル記述子をシャドウプロセスに渡して標準 C ライブラリの close 関数を実行する．

4.2.2　ソケット

TA が UNIX ドメインソケットまたはネットワークソケットを用いて CA との通信を行うために実装した POSIX API の一覧を**表 2** に示す．TA がソケットを作成する際には専用ライブラリの socket 関数が呼び出される．この関数には引数で UNIX ドメインソケットかネットワークソケットを指定し，その情報を RPC 用バッファに格納してシャドウプロセスを呼び出す．シャドウプロセスは標準 C ライブラリの socket 関数を呼び出し，ノーマルワールドにソケットを作成する．その後，ファイル記述子を専用ライブラリ経由で TA に返り値として返す．

TA がクライアントになる場合，CA として動作するサーバに接続する際に専用ライブラリの connect 関数が呼び出される．この関数は引数の情報を RPC 用バッファに格納してシャドウプロセスを呼び出し，標準の connect 関数を実行する．TA がサーバになる場合，クライアントからの接続要求を待つために専用ライブラリの bind 関数，listen 関数，accept 関数が呼び出される．これらの関数も引数の情報を RPC 用バッファに格納してシャドウプロセスを呼び出し，対応する標準のソケット関数を実行する．ノーマルワールドで accept 関数を実行してクライアントからの接続要求を待つ間，TA は RPC の完了を待って停止する．

表 2　ソケットのための POSIX API

関数	動作
socket	新しいソケットを作成
bind	ソケットにアドレスを関連付ける
listen	接続待ちの状態にソケットを移行
accept	接続を受け入れる
connect	ソケットに接続する
send	指定されたソケットにデータを送信
recv	指定されたソケットからデータを受信

接続が確立した後，TA がデータを送信する際には専用ライブラリの send 関数が呼び出される．この関数は引数のデータを RPC 用バッファにコピーし，シャドウプロセスを呼び出して標準の send 関数を実行する．TA がデータを受信する際には専用ライブラリの recv 関数が呼び出される．この関数はシャドウプロセスを呼び出して標準の recv 関数を実行し，受信したデータを RPC 用バッファから TA のバッファにコピーする．ノーマルワールドで CA からのデータが受信できるまでの間，TA は RPC の完了を待って停止する．

4.3　CA に提供する専用ライブラリ

CA と TA を並列に実行できるようにするために，TZmediator は CA にも専用ライブラリを提供する．通常，CA が TA のコマンドを呼び出した後，TA においてコマンドの実行が終了するまで CA は停止し続ける．TA が長時間動き続ける場合やデータの受信を長時間待つ場合，CA を並列して実行することはできない．そのため，この専用ライブラリは CA の実行開始時にサブスレッドを生成し，サブスレッドに TA の開始コマンドを呼び出させる．メインスレッドを用いてクラウドアプリケーションの実行を行うことで，TA の実行中にも CA を並列に実行することができる．

この専用ライブラリは TEE Client API v1.0 に準拠しており，生成したサブスレッドが 4.2 節に示した TA の必須エントリポイントに対応する関数を呼び出す．これにより，CA として動作することを意識せずにクラウドアプリケーションを開発できるようになる．まず，TA のインスタンスを作成するために TEEC_InitializeContext 関数を呼び出す．次に，新しいセッションを開くために TA の UUID を指定

© 1959 Information Processing Society of Japan

して TEEC_OpenSession 関数を呼び出す．その後，TA の開始コマンドを実行するために TEEC_InvokeCommand 関数を呼び出す．開始コマンドの実行が終了すると，セッションを閉じるために TEEC_CloseSession 関数を呼び出す．最後に，TA のインスタンスを破棄するために TEEC_FinalizeContext 関数を呼び出す．

5. 実験

TZmediator を用いて CA と TA が双方向に通信できることを確認し，通信性能とオーバヘッドを測定する実験を行った．通信に用いた POSIX API は名前付きパイプ，UNIX ドメインソケット，ネットワークソケットの 3 つであった．ネットワークソケットを用いる場合には TCP/IP を用いた．比較として，OP-TEE の専用 API を用いた CA と TA 間の通信についても性能を測定した．この通信は CA と TA 間に確保された共有メモリにデータを格納し，TA のコマンドを呼び出すことで行われる．TZmediator はまだ WASI に対応できていないため，本実験では Wasm を用いずに TA を実行した．

本実験は QEMU 8.0.0 [15] を用いて ARM Cortex-A57 をエミュレーションすることにより行った．QEMU には 1057MB のメモリを割り当て，OP-TEE 4.0.0 を動作させた．TrustZone のノーマルワールドでは Linux 6.0.0 を動作させた．実験に用いたマシンの CPU は Intel Core i7-12700，メモリは 64GB，OS は Linux 6.2.0 であった．

性能測定の際には，セキュアワールドでもノーマルワールドと同じくナノ秒単位で時刻を取得した．OP-TEE においては，セキュアワールド内の時間分解能はミリ秒単位となっている．ノーマルワールドで動作する Linux カーネルの単調クロックが提供する時刻と同じ時刻を取得できるようにするために，TEE Internal Core API の TEE_Time 構造体を拡張し，OP-TEE ドライバを修正した．また，セキュアワールドで TEE_GetREETime 関数を用いて時刻を取得するのにかかる時間は，ノーマルワールドへの遷移を伴うため 77 マイクロ秒であった．そのため，この時間を考慮して経過時間を算出した．

5.1 POSIX API の動作確認

CA と TA 間で POSIX API を用いて通信が行えることを確認する実験を行った．名前付きパイプを用いる場合，CA が名前付きパイプを作成して文字列を書き込み，TA がそのデータを読み込んで出力した．また，TA が名前付きパイプを作成して文字列を書き込み，CA がその文字列を読み込んで出力した．ソケットを用いる場合，CA が TA に接続して文字列を送信し，TA がその文字列を受信して出力した．また，TA が CA に接続して文字列を送信し，CA がその文字列を受信して出力した．TCP ソケットを用いる場合には，接続先のアドレスに 127.0.0.1 (localhost) を指定した．その結果，名前付きパイプ，UNIX ドメインソケット，TCP ソケットの 3 種類の POSIX API について，CA と TA の両方で受け取った文字列が正常に出力されることを確認した．

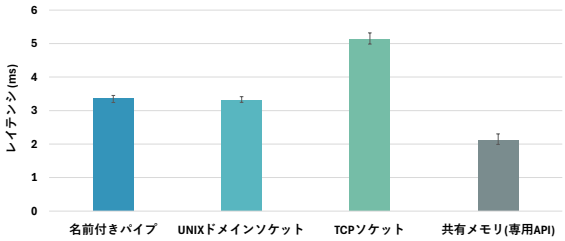

図 5 通信の往復レイテンシ

5.2 通信の往復レイテンシ

POSIX API を用いて CA と TA 間で 4KB のデータを送受信する往復レイテンシを測定した．まず，CA がパイプやソケットにデータを書き込み，シャドウプロセス経由で TA がそのデータを読み込む．その後，TA が同じパイプやソケットにデータを書き込み，シャドウプロセス経由で CA がそのデータを読み込む．比較として専用 API を用いる場合には，CA が共有メモリにデータを格納してから TA のコマンドを呼び出す．そして，TA が共有メモリにデータを格納してからコマンド実行を終了して CA に戻る．CA がデータの書き込みを開始してから読み込みを終了するまでの時間を計測した．

測定結果を**図 5** に示す．TZmediator は専用 API を用いたデータの受け渡しよりも 1.6〜2.4 倍の実行時間がかかることが分かった．これはセキュアワールド内の TA がデータを読み書きする際に，ノーマルワールド内のシャドウプロセスを RPC で呼び出してデータの転送を行うためである．TCP ソケットを用いた通信は，TCP/IP プロトコルスタックを経由することでオーバヘッドが大きくなり，ばらつきも大きくなったため，名前付きパイプや UNIX ドメインソケットを用いた通信よりも平均が長くなったと考えられる．

次に，バッファサイズを 1 バイトから 8KB まで変更して通信の往復レイテンシを測定した．測定結果を**図 6** に示す．256 バイトの場合と比べて，8KB の場合の往復レイテンシは名前付きパイプで 11%，UNIX ドメインソケットで 6.5%，TCP ソケットで 7.2%長くなることが分かった．この性能低下の原因は，データの書き込みの際にシャドウプロセスに渡す構造体にデータをコピーすることや，読み込みの際に専用ライブラリが TA から渡されたバッファにデータをコピーすることに起因している．

5.3 通信のスループット

CA と TA 間で POSIX API を用いて 1 秒あたりに送信で

© 1959 Information Processing Society of Japan

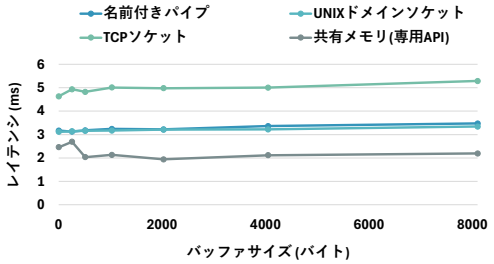

図 6 通信の往復レイテンシ（様々なバッファサイズ）

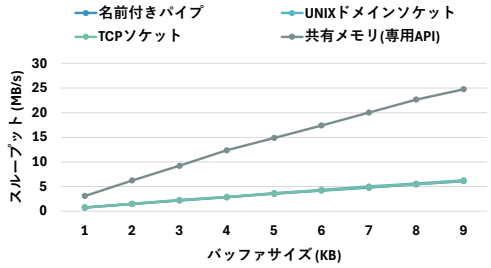

図 7 CA と TA 間の通信のスループット

きるデータ量を測定した．名前付きパイプでは write 関数，UNIX ドメインソケットおよび TCP ソケットでは send 関数を用いてそれぞれ 1000 回の送信を行い，送信を開始してから受信を完了するまでにかかった時間からスループットを算出した．この実験では，CA から TA へ送信する場合について，送信するバッファサイズを変えながらのスループットを測定した．比較として，専用 API を用いて CA から TA に共有メモリ経由でデータを送信した場合のスループットも測定した．

測定結果を**図 7** に示す．TZmediator は専用 API を用いた場合と比較して，スループットが 75〜78%低下することが分かった．この性能低下もセキュアワールド内の TA がデータを読み込む際に，ノーマルワールド内のシャドウプロセスを RPC で呼び出すことによるオーバヘッドが生じたためである．

5.4 オーバヘッドの内訳

TZmediator のオーバヘッドの内訳を調べるために，TA がシャドウプロセスとやりとりするのにかかる時間を測定した．この実験では，セキュアワールド内の TA が RPC を用いてノーマルワールド内のシャドウプロセスを呼び出すのにかかる時間と，シャドウプロセスから TA に戻ってくるのにかかる時間を調べた．RPC で渡すデータサイズは 0 バイトとした．比較として，ノーマルワールド内の CA が TA のコマンドを呼び出すのにかかる時間と，コマンドを終了して TA から CA に戻ってくるのにかかる時間を調べた．

測定結果を**図 8** に示す．この結果より，RPC でシャドウプロセスを呼び出して戻ってくるだけで 1.6 ミリ秒かかることが分かった．ただし，実際の通信では RPC の一部と CA やシャドウプロセスの処理がオーバラップするため，通信時間に占める割合はこれよりも小さくなる．また，TA による RPC の呼び出しは CA によるコマンドの呼び出しより 17%高速であった．また，RPC の呼び出しから戻る時間はコマンドから戻る時間より 35%高速であった．

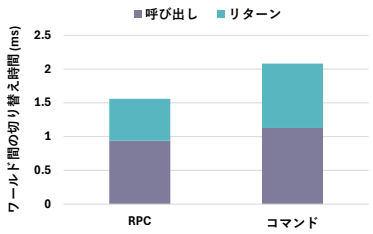

図 8 RPC の実行時間

6. 関連研究

Intel SGX はプロセス内にエンクレイヴを作成し，その中でプログラムを安全に実行することを可能にする．エンクレイヴ内のプログラムに POSIX API を提供するシステムがいくつか提案されている．SCONE [16] はエンクレイヴ内のプログラムに専用の C ライブラリを提供する．このライブラリは非同期システムコールのためのインタフェースを用いてホスト OS を呼び出す．例えば，ネットワークソケットのための socket 関数や connect 関数などをサポートしている．

信頼できるアプリケーションの間で POSIX API を用いた通信が行えるシステムも提案されている．Graphene-SGX [17] はエンクレイヴ間で POSIX API を用いて通信を行うためのライブラリ OS を提供しており，複数のエンクレイヴが協調して動作することができる．通常のプロセスと同様に fork 関数や exec 関数などを用いることができ，エンクレイヴ外部の PAL を呼び出して RPC を用いることでパイプなどのプロセス間通信を実現している．Occlum [18] は POSIX API を用いるためのライブラリ OS を提供しており，エンクレイヴの内部で複数のプロセスを実行することができる．そのため，プロセス間通信はエンクレイヴ内のライブラリ OS 経由で効率よく行うことができる．

WaTZ [12] は TrustZone のセキュアワールドで Wasm アプリケーションを TA として実行することを可能にしている．さらに，その Wasm アプリケーションの軽量なリモートアテステーションも提供しており，セキュアワールド内で実行されるコードの正当性を検証することができる．WaTZ は Wasm のシステムインタフェースである WASI の一部を TEE Internal Core API にマッピングすることで，Wasm アプリケーションがノーマルワールド経由でリモートホストとの通信を行うことができる．TZmediator はノーマルワールド内の CA との様々な通信を可能にして

© 1959 Information Processing Society of Japan

いる点が異なる．

Wasm のシステムインタフェースの WASI は現在，標準化が行われているところである．WASI Preview 1 は POSIX API に似た API を提供しているが，ソケットを十分にサポートできていない．WASI Preview 2 ではソケットのサポートも強化されている．WASI 以外のシステムインタフェースとして，WASI を拡張して POSIX 対応にした WASIX [19] が提案されている．また，WebAssembly Linux Interface (WALI) [20] は Linux のシステムコールのインタフェースを提供している．WALI を用いることで Wasm で Linux アプリケーションを動作させることができる．

ReZone [8] は TrustZone のセキュアワールドを分割して，それぞれの TA を隔離実行することを可能にしている．TA ごとにゾーンと呼ばれるサンドボックスが作成され，個別の Trusted OS が提供される．ゾーンは専用のリソースのみにアクセスすることができ，他のゾーンやノーマルワールドへのアクセスが制限される．ReZone は TrustZone とは別に既存のハードウェアを活用することで実現されている．ReZone ではセキュアワールドのリソースが分割されるため，クラウドアプリケーションに利用可能なリソースが限られる可能性がある．

7. まとめ

本稿では，クラウドアプリケーションを 2 つのワールドに分割し，ワールド間で POSIX API を用いて協調実行を行うシステム TZmediator を提案した．TZmediator は保護する必要がある処理のみをセキュアワールドで TA として実行し，それ以外の処理はノーマルワールドで CA として実行する．ワールド間にまたがって POSIX API を用いた通信を行えるようにするために，ノーマルワールド内に TA に対応するシャドウプロセスを作成する．CA と TA は並列に実行され，シャドウプロセスを介して通信を行う．現在のところ，名前付きパイプ，UNIX ドメインソケット，ネットワークソケットに対応している．実験により，CA と TA 間の通信性能を確認した．

今後の課題は，TA とシャドウプロセス間の RPC を高速化することである．通信データのコピーを削減したり，RPC の呼び出し方法を改善することを検討している．また，セキュアワールドで TA として実行するプログラムを Wasm を用いてより安全に実行できるようにすることも考えている．Wasm アプリケーションを動かすことはできているため，WASI 等を用いて CA と通信が行えるようにする予定である．さらに，名前付きパイプやソケット以外の通信にも対応する必要もある．

謝辞 本研究の一部は，JST，CREST，JPMJCR21M4 の支援を受けたものである．また，本研究の一部は，国立研究開発法人情報通信研究機構の委託研究（05501）による成果を含む．

参考文献

[1] Intel: Intel Software Guard Extensions (Intel SGX), https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html.
[2] AMD: AMD Secure Encrypted Virtualization (SEV), https://www.amd.com/ja/developer/sev.html.
[3] ARM: Cortex-A 用 TrustZone，https://www.arm.com/ja/technologies/trustzone-for-cortex-a.
[4] Keystone: An Open Framework for Architecting TEEs, https://keystone-enclave.org/.
[5] Trusted Firmware: OP-TEE, https://www.trustedfirmware.org/projects/op-tee.
[6] Open-TEE: Open-TEE, https://open-tee.github.io/.
[7] Google: Trusty TEE — Android Open Source Project, https://source.android.com/docs/security/features/trusty?hl=ja.
[8] Cerdeira, D., Martins, J., Santos, N. and Pinto, S.: ReZone: Disarming TrustZone with TEE Privilege Reduction, *31st USENIX Security Symposium*, pp. 2261–2279 (2022).
[9] GlobalPlatform: GlobalPlatform Homepage, https://globalplatform.org/.
[10] GlobalPlatform: *GlobalPlatform Technology TEE Internal Core API Specification Version 1.3.1* (2021).
[11] GlobalPlatform: *GlobalPlatform Device Technology TEE Client API Specification Version 1.0* (2010).
[12] Menetrey, J., Pasin, M., Felber, P. and Schiavoni, V.: WaTZ: A Trusted WebAssembly Runtime Environment with Remote Attestation for TrustZone, *2022 IEEE 42nd International Conference on Distributed Computing Systems* (2022).
[13] OP-TEE: optee_os at 4.0.0, https://github.com/OP-TEE/optee_os/tree/4.0.0.
[14] OP-TEE: Trusted Applications, https://optee.readthedocs.io/en/latest/building/trusted_applications.html.
[15] QEMU: QEMU - A generic and open source machine emulator and virtualizer, https://www.qemu.org/.
[16] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P. and Fetzer, C.: SCONE: Secure Linux Containers with Intel SGX, *12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 689–703 (2016).
[17] Tsai, C., Porter, D. E. and Vij, M.: Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX, *2017 USENIX Annual Technical Conference*, pp. 645–658 (2017).
[18] Shen, Y., Tian, H., Chen, Y., Chen, K., Wang, R., Xu, Y., Xia, Y. and Yan, S.: Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX, *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020).
[19] WASIX: WASIX - The Superset of WASI, https://wasix.org/.
[20] Ramesh, A., Huang, T., Titzer, B. L. and Rowe, A.: Stop Hiding The Sharp Knives: The WebAssembly Linux Interface, *arXiv.org e-Print archive, arXiv:2312.03858v1* (2023).

© 1959 Information Processing Society of Japan