Arm TrustZoneにおけるPOSIX APIを用いたクラウドアプリケーションの協調実行

佐藤 太陽[1]　光来 健一[1]

概要：エッジコンピューティングにより，クラウドはアプリケーションをユーザの近くで実行できるようになる．信頼できないエッジデバイスであっても Arm TrustZone を用いることにより，セキュアワールドで TA としてクラウドアプリケーションを安全に実行できる．しかし，セキュアワールドは高い権限を持つため，クラウドアプリケーションに脆弱性が存在すると，システム全体に影響を及ぼす恐れがある．そのため，セキュリティを保つ必要のない処理はノーマルワールドで CA として実行することが望ましいが，CA と TA 間の連携には専用 API を使用する必要があり，柔軟な協調は容易ではない．本稿では，2 つのワールドに分割されたクラウドアプリケーションに対して，POSIX API を用いて協調実行を行うことを可能にする TZmediator を提案する．TZmediator はノーマルワールドに各 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．また，より強力な隔離が要求される場合には WebAssembly を用いて TA を実行することができる．OP-TEE を用いて TZmediator を実装し，CA と TA 間の通信性能を調べた．

1. はじめに

近年，クラウドアプリケーションをエンドユーザの近くで実行することで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 [1]．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 [2] が，クラウドアプリケーションを安全に実行するために，CPU が提供する Trusted Execution Environment (TEE) を利用することができる．TEE は OS を含むシステムの他の部分からメモリを隔離し，攻撃者が TEE で使用されるコードやデータにアクセスできないようにする，TEE のメモリが暗号化される場合には物理的な盗聴も防ぐことができる．

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ており，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全体をセキュアワールドで実行することも可能であるが，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 [3]．そのため，TA として実行されるコードを最小限に抑えて，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 の協調実行には TEE 用に定義された GlobalPlatform API [4], [5] を使用する必要があり，CA と TA 間の柔軟な協調は容易ではない．

そこで本稿では，2 つのワールドに分割されたクラウドアプリケーションに対して，POSIX API [6] を用いて協調実行を行うことを可能にする 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．POSIX API を介してワールド間のシームレスな通信を実現するために，TZmediator はノーマルワールド内に各 TA に対応するシャドウプロセスを作成する．TA が POSIX API を呼び出す際には，その呼び出しをシャドウプロセスに転送する．より強力な分離を必要とする場合は TA 内で WebAssembly アプリケーションを実行し，拡張された WebAssembly System Interface (WASI) を介して CA と通信できるようにする．

TZmediator を OP-TEE に実装し，TA と CA にそれぞれ提供される TZm ライブラリを開発した．TZm-TA ライブラリは，TA が CA と通信できるように POSIX API を提供する．呼び出された POSIX API は軽量な遠隔手続き呼び出し (RPC) またはポーリングベースの方式を使用してシャドウプロセスに転送される．現在，パイプやソケットなどのメッセージパッシング API や共有メモリ API，シグナル API をサポートしている．TZm-CA ライブラリは，

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

CA に posix_spawn 関数を提供し，TA をシームレスに生成してシャドウプロセスを作成する．

TZmediator を用いて CA と TA 間の通信性能を調べる実験を行った．その結果，ノーマルワールド内での通信と比較して，TZmediator はほとんどの POSIX API において同等の性能を達成した．さらに，POSIX API を用いることで，実際のクラウドアプリケーションをわずかなオーバヘッドで実行できることも確認した．

以下，2 章で TrustZone を用いてクラウドアプリケーションを安全に実行する際の問題について述べる．3 章で POSIX API を用いて 2 つの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 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

エッジコンピューティングはエンドユーザに近い場所でクラウドアプリケーションを実行する [2]．そのため，集中型であるクラウドコンピューティングと比較して，低遅延・高速・リアルタイム処理が可能になる．また，クラウドとの通信量を削減し，ネットワーク負荷を軽減することもできる．しかし，エッジデバイスはクラウドの制御外にあるため必ずしも信頼できるとは限らない．適切に保守されていない場合は機密情報の漏洩やアプリケーションの改竄，リバースエンジニアリングなどの攻撃を受けやすくなる可能性がある．加えて，一般的なエッジデバイスはリソースが限られているため，強力なセキュリティ機構を実装することが難しい．

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 が提供する Trusted Execution Environment (TEE) を用いることが考えられる．TEE は論理的に分離されたメモリ領域内でのプログラムの実行を可能にする．TEE 外部のシステムは，TEE で使用されるコードやデータの盗聴や改竄ができない．分離されたメモリが CPU によって暗号化されている場合は物理メモリの盗聴を防ぐことができる．主要な CPU で実装されている TEE アーキテクチャの例としては，アプリケーション向けの Intel SGX [7]，Arm TrustZone [8]，RISC-V Keystone [9]，仮想マシン向けの AMD SEV [10]，Intel TDX [11]，Arm CCA [12] などがある．TEE を用いることで，OS やハイパーバイザが侵害された場合でもエッジデバイス上のクラウドアプリケーションを保護できる．

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 として機能し，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．OP-TEE [13] や Open-TEE [14]，Trusty [15] などの Trusted OS もセキュアワールド内で実行され，TA の実行をサポートする．一方，ノーマルワールドは従来の実行環境として機能し，Rich Execution Environment (REE) とも呼ばれる．従来のアプリケーションだけでなく，TA を利用するための Client Application (CA) も実行する．Linux などの Rich OS はこれらのアプリケーションをサポートするためにノーマルワールドで実行される．これらの OS の下で実行されるセキュアモニタは 2 つのワールド間の切り替えを管理する．システムリソースは厳密に分離されているため，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も高い権限を持つことである．エッジデバイスで実行されるクラウドアプリケーションは従来の TA よりも複雑な場合が多いため，脆弱性を含む可能性が高くなる．ノーマルワールドに侵入した攻撃者がこのような脆弱性を悪用すると，権限を昇格してセキュアワールドを制御できるようになる可能性がある [3]．もう 1 つは，複数の TA がセキュアワールド内，つまり単一の実行環境で実行されることである．侵害されたクラウドアプリケーションは他の TA を攻撃し，それらが保持する機密情報を窃取する可能性がある．

クラウドアプリケーションのセキュリティを強化するには，DarkneTZ [16] のようにセキュアワールドで TA として実行されるコードを最小限に抑えることが望ましい．この設計では，TA は機密情報のみを保持し，それを処理するコードのみを実行する．CA はセキュリティに影響しないその他のタスクを実行し，必要に応じて TA と通信する．しかし，TrustZone では CA と TA 間の協調実行には，GlobalPlatform によって定義された専用の API，具体的には CA に提供される TEE Client API [4] と TA に提供される TEE Internal Core API [5] を使用する必要がある．これらの API は標準化されているものの，Rich OS で一般的に使用されるパイプやソケットなどの POSIX API とは大きく異なる．例えば，TA は CA によるコマンド呼び出しによってのみ実行され，CA は TA の実行が完了するまで待機する必要がある．その結果，CA と TA 間の柔軟な連携は困難である．このため，2 つのワールドに分割されたクラウドアプリケーションの開発はより困難になる．

本稿における脅威モデルは以下の通りである．TrustZone 機能を提供するハードウェアやセキュアモニタには脆弱性がなく，TrustZone が提供する保護機構は侵害されないと仮定する．セキュアワールドで実行されるソフトウェアは信頼するが，脆弱性のない TA だけでなく，脆弱性のある TA もクラウドアプリケーションに含まれる場合があることを想定する．一方，ノーマルワールドで実行されるソフトウェアは Rich OS を含めて信頼しない．したがっ

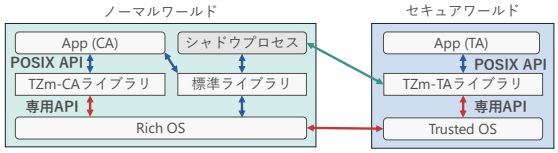


図 1: TZmediator のシステム構成

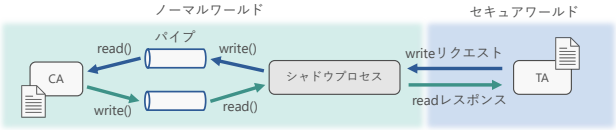


図 2: パイプを用いた通信の例

て，クラウドアプリケーションの一部である CA は侵害される可能性があると仮定する．本稿では，クラウドアプリケーションで扱われる機密情報の盗聴や改竄を行おうとするノーマルワールド内の攻撃者を対象とし，サービス妨害 (DoS) 攻撃や物理的な攻撃などのその他の攻撃は対象としない．

3. TZmediator

本稿では，TrustZone の 2 つのワールドに分割されたクラウドアプリケーションにおいて POSIX API を用いた協調実行を可能にする 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．POSIX API を用いて CA と TA 間のシームレスなワールド間通信を可能にするため，TZmediator は各 TA に対しノーマルワールドにシャドウプロセスを作成する．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．ライブラリとシステムコールのセマンティクスは CA で使用されるものと同じであるため，POSIX との高い互換性を実現することができる．プロセス間通信用の POSIX API の例としては，パイプやソケットなどのメッセージパッシング API や共有メモリ API，シグナル API などが挙げられる．

POSIX API をエミュレートするために，TZmediator は TA に TZm-TA ライブラリを提供する．このライブラリは，`pipe` や `socket` といった POSIX API の通信に必要な関数を提供する．TA が TZm-TA ライブラリを呼び出すと，このライブラリはノーマルワールド内の対応するシャドウプロセスと通信する．シャドウプロセスは，TA に代わって標準ライブラリを用いて指定された POSIX API を実行し，結果を TA に返す．一方，CA は Rich OS が提供する標準ライブラリを介して POSIX API を使用する．CA は異なるワールドで動作する TA と直接通信する代わりに，対応するシャドウプロセスと通信し，シャドウプロセスが TA と通信する．

例として，TA がパイプを用いて CA と通信する際の処理の流れを図 2 に示す．TA がパイプにデータを書き込む場合，TZm-TA ライブラリはシャドウプロセスに書き込み要求を送信し，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 がデータを読み込む場合，TZm-TA ライブラリはシャドウプロセスに読み込み要求を送信し，シャドウプロセスが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．

さらに，TZmediator は CA と TA を GlobalPlatform API を使用しない通常のプロセスとして作成することを可能にする．これにより，ユーザはクラウドアプリケーションを通常のプロセスの集合として開発できる．そのために，TZmediator は CA に TZm-CA ライブラリを提供する．CA はこのライブラリが提供する `posix_spawn` 関数を用いて新しい TA を生成する．この関数は TEE Client API を用いた TA に関する複雑な処理を CA から隠蔽する．この時，TA のコマンドはサブスレッドを作成することで CA と並行して実行される．同様に，TA に提供される TZm-TA ライブラリは TEE Internal Core API を用いた処理を TA から隠蔽する．TZm-TA ライブラリは TZm-CA ライブラリからのコマンドを受け取り，クラウドアプリケーションで定義された関数を実行する．

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る場合には WebAssembly (Wasm) を用いて TA を実行する [17]．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 を攻撃することは難しい．Wasm アプリケーションは POSIX に似た API を提供する WebAssembly System Interface (WASI) [18] を用いる．TZmediator は WASI を拡張して POSIX との互換性を高める．TA 内の Wasm アプリケーションが WASI ライブラリによって提供される関数を呼び出すと，同じ TA 内の Wasm ランタイムは TZm-TA ライブラリを介して対応するシャドウプロセスを呼び出す．

4. 実装

TZmediator を OP-TEE 3.15.0 [13] に実装した．OP-TEE は TA の実行に必要な OP-TEE OS や tee-supplicant などを提供する．OP-TEE OS はセキュアワールドで動作する Trusted OS であり，tee-supplicant はノーマルワールドで動作して TA からのリクエストを処理するデーモンである．また，OP-TEE は GlobalPlatform API に準拠したライブラリも提供する．ノーマルワールドで動作する Rich OS には OP-TEE ドライバを含む Linux を用いた．TA で

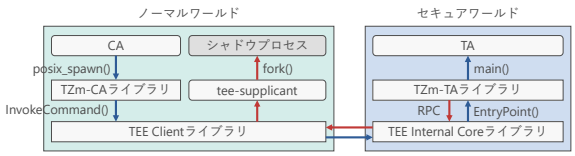


図 3: TA とシャドウプロセスの生成

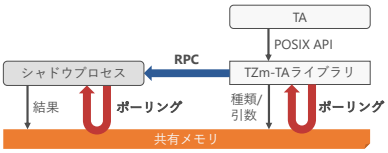


図 4: TA からシャドウプロセスへの POSIX API の転送

Wasm アプリケーションを実行するために，WebAssembly Micro Runtime (WAMR) [19] と WASI Preview 1 [18] を拡張した．

4.1 TA の生成

TZmediator は TZm-CA ライブラリでエミュレートされた `posix_spawn` 関数を提供し，CA が TA を子プロセスのように生成する．図 3 に TA を生成する処理の流れを示す．標準ライブラリが提供する `posix_spawn` 関数は引数として実行ファイルのパス名を受け取るが，CA では代わりに TA に割り当てられた UUID の文字列を指定する．通常のパス名が指定された場合には，TZm-CA ライブラリは標準ライブラリで提供される従来の `posix_spawn` 関数を呼び出す．

TZm-CA ライブラリの `posix_spawn` 関数は TEE Client API で定義されている関数を呼び出して TA を実行するため，クラウドアプリケーションでこれらの関数の呼び出しを行う必要はない．CA を TA と並行して実行するために，TZm-CA ライブラリはコマンド呼び出しの前にサブスレッドを作成し，メインスレッドでクラウドアプリケーションの実行を継続する．また，TA に提供される TZm-TA ライブラリにおいて TEE Internal Core API で定義されている必須エントリポイント [20] を実装しているため，クラウドアプリケーションがそれらを実装する必要はない．

TA で Wasm アプリケーションを生成する場合，CA は Wasm アプリケーションのパス名を指定する．`posix_spawn` 関数は Wasm ランタイムを TA として暗黙的に実行する．セキュアワールドでは TZm-TA ライブラリが Wasm ランタイムを実行し，AOT (Ahead-of-Time) コンパイラを使用してネイティブコードにコンパイルされた Wasm アプリケーションをロードして `main` 関数を実行する．

4.2 シャドウプロセス

TZm-TA ライブラリは新しいセッションが確立されると，ノーマルワールドの TA に対応するシャドウプロセスを作成する．まず，ノーマルワールドの tee-supplicant に RPC 要求を送信する．次に，tee-supplicant のプラグインはシャドウプロセス用の子プロセスを作成し，そのプロセス ID を TA に返す．tee-supplicant はシャドウプロセス用のプラグインが提供するハンドラを使用して RPC 要求を処理する．tee-supplicant のプラグインは，TEE Client API を使用して共有ライブラリとして実装され，tee-supplicant の起動時に動的にロードされる．プラグインには UUID が割り当てられ，TA は RPC ハンドラを指定するために使用する．RPC 要求を受信すると，tee-supplicant は指定された UUID に基づいてプラグインに要求をディスパッチする．

4.3 POSIX API の転送

セキュアワールドで実行される TA からノーマルワールド内のシャドウプロセスに POSIX API の呼び出しを転送するために，TZmediator は TA と tee-supplicant 間で用いられている RPC を利用できる．しかし，その場合では共有メモリの割り当て，POSIX API の実行，共有メモリの解放の各段階で RPC を発行する必要があり，合計 6 回のワールド切り替えが発生するため，オーバヘッドが大きい．

この問題を解決するために，TZmediator は 2 回のワールド切り替えのみを必要とする軽量 RPC を提供する．図 4 に示すように，TA とシャドウプロセス間で共有メモリを事前に割り当て，すべての POSIX API 呼び出しで再利用する．CA が TA を生成する際に，一意の名前を指定して共有メモリオブジェクトを作成し，両者でマッピングする．TA が POSIX API の関数を実行すると，TZm-TA ライブラリは関数の種類と引数を共有メモリに作成されたバッファに格納する．次に，コンテキストをノーマルワールドに切り替え，対応するシャドウプロセスを起こす．このスレッドはシャドウプロセスが POSIX API の実行を完了するまで待機する．セキュアワールドに戻ってくると，TA は共有バッファから POSIX API の実行結果を受け取る．

さらに，ワールド切り替えを行わずに済ませるために，TZmediator は共有メモリを用いたポーリングベースの方式もサポートする．シャドウプロセスと TZm-TA ライブラリは共有バッファをポーリングし，リクエストの送信と POSIX API の実行完了を検知する．CPU オーバヘッドを抑制するために，TA とシャドウプロセスは共有バッファの確認とマイクロ秒単位の短時間のスリープを繰り返すことができる．

4.4 サポートする POSIX API

TZmediator はプロセス間の通信に用いる様々な POSIX API をサポートしている．

4.4.1 パイプ

TA は名前付きパイプ (FIFO) を作成する際に，TZm-TA

ライブラリが提供する mkfifo 関数を呼び出す．この関数は引数で指定されたパス名とアクセスモードをシャドウプロセスとの共有バッファに格納し，軽量 RPC またはポーリングベースの方式でシャドウプロセスを呼び出す．その後，シャドウプロセスは受信したパス名とアクセスモードを使用して標準ライブラリの mkfifo 関数を呼び出して名前付きパイプを作成する．そのため，TA はノーマルワールドで有効なパス名を指定する必要がある．シャドウプロセスは結果を共有バッファに格納し，TZm-TA ライブラリを介して TA に結果を返す．名前付きパイプを削除するには，TA はシャドウプロセスを介してノーマルワールドで unlink 関数を実行する．

名前付きパイプを開くには TA はパス名を指定して open 関数を呼び出す．mkfifo 関数と同様に，この関数は TA に代わってシャドウプロセスによって実行され，結果のファイル記述子を TA に返す．write 関数を使用してパイプにバイトストリームを送信すると，TZm-TA ライブラリは指定されたデータを共有バッファにコピーし，シャドウプロセスを呼び出す．シャドウプロセスは共有バッファ内のデータをファイル記述子で指定された名前付きパイプに書き込む．名前付きパイプを閉じるには TA はファイル記述子を指定して close 関数を呼び出す．

CA が名前なしパイプを使用して TA と通信する際には，まず標準ライブラリの pipe 関数を呼び出して，読み取り用および書き込み用のパイプのペアを作成する．次に，この関数によって返されたファイル記述子のペアを posix_spawn 関数の引数として指定し，新たに生成される TA と共有する．TZm-CA ライブラリの posix_spawn 関数は TA の生成時に，TA が作成するシャドウプロセスへファイル記述子を引き渡す．この手法はファイル記述子パッシングと呼ばれ，UNIX ドメインソケットを介してプロセス間でファイル記述子を共有する．

4.4.2 ソケット

TA はソケットを作成する際に，TZm-TA ライブラリの socket 関数を呼び出す．この関数は UNIX ドメインの場合は AF_UNIX，TCP または UDP の場合は AF_INET などのアドレスファミリを受け取る．シャドウプロセスは標準の socket 関数を呼び出してノーマルワールドにソケットを作成し，生成されたファイル記述子を TA に返す．

TA がクライアントとして CA に接続する場合，TZm-TA ライブラリの connect 関数を呼び出す．UNIX ドメインソケットの場合は CA が使用するパス名を引数として指定し，TCP ソケットの場合は TA と CA は同じホスト上で動作するため，ループバックアドレスと一意のポートを指定する．TA がサーバとして CA からの接続を待機する場合，TZm-TA ライブラリが提供する bind 関数や listen 関数，accept 関数を呼び出す．接続が確立されると，TA は TZm-TA ライブラリの send 関数や recv 関数を使用してデータを送受信する．UDP ソケットの場合，TA は接続を確立せずにループバックアドレスを指定して，sendto 関数と recvfrom 関数を使用する．

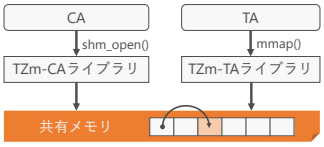


図 5: POSIX 共有メモリのエミュレーション

4.4.3 共有メモリ

CA と TA 間の共有メモリをサポートするために，TZmediator は図 5 に示すように，共有メモリ用の POSIX API をエミュレートする．CA が TA を生成すると，TZm-CA ライブラリは十分なサイズの共有メモリを事前に割り当てて登録し，POSIX 共有メモリオブジェクトを管理するためのデータ構造を作成する．TZm-CA ライブラリと TZm-TA ライブラリはどちらも，POSIX 共有メモリを操作するためのエミュレートされた関数群を提供する．CA または TA が名前を指定して shm_open 関数を呼び出すと，引数で指定した名前の新しい共有メモリオブジェクトを管理データ構造に登録し，ファイル記述子を返す．ftruncate 関数は指定された共有メモリオブジェクトのために，事前に割り当てられた共有メモリ内に指定されたサイズの新しいメモリ領域を割り当てる．mmap 関数は割り当てられたメモリ領域のアドレスを返す．

4.4.4 シグナル

TZmediator はシャドウプロセスと TZm-TA ライブラリにおけるシグナル API をエミュレートする．TA はシグナルを送信する際に TZm-TA ライブラリが提供する kill 関数を呼び出す．TZm-TA ライブラリは指定されたシグナルを Rich OS 経由で送信するために，シャドウプロセスを呼び出す．TA がシグナルを受信できるようにするために，シャドウプロセスは TA に向けて送信されたシグナルを代理で受信する．そのために，シャドウプロセスが生成される時に捕捉可能なすべてのシグナルに対してシグナルハンドラを登録する．図 6 に示すように，シグナルハンドラが呼び出されると，受信したシグナル番号を TA とシャドウプロセス間の共有メモリ内に作成されたシグナルキューに格納する．TZm-TA ライブラリは提供する関数が呼び出される時などに，シグナルキューを定期的に確認する．シグナル番号がキューに格納されている場合，TZm-TA ライブラリはそれをキューから取り出し，対応するシグナルハンドラを呼び出す．TA がそのシグナルに対して signal 関数や sigaction 関数を用いてカスタムハンドラを登録している場合にはデフォルトのアクションではなく，そのハンドラを実行する．シグナルハンドラの登録時にはシャドウプロセスを呼び出すのではなく，TZm-TA ライブラリ内にシグナルハンドラを登録する．

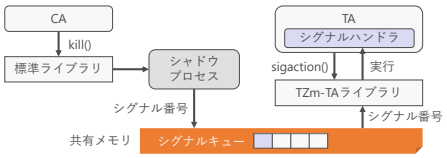


図 6: シグナル受信のエミュレーション

4.5 WASI の拡張

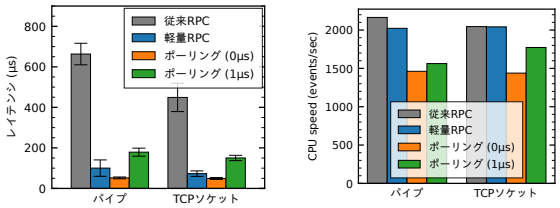
TZmediator は WASI を拡張し，Wasm アプリケーションが通信関連の POSIX API を使用できるようにする．TA 内で実行されている Wasm アプリケーションが WASI ライブラリによって提供される関数を呼び出すと，その関数は同じ TA 内で実行されている Wasm ランタイムを呼び出す．次に，Wasm ランタイムは TZm-TA ライブラリを呼び出し，TZm-TA ライブラリが最終的にシャドウプロセスを呼び出す．このように，TA 内の Wasm アプリケーションはネイティブアプリケーションと同様に CA と通信できる．

TZmediator は Wasm アプリケーションが POSIX API 経由で POSIX 共有メモリにアクセスできるようにする．Wasm ではメモリを単一の連続したバイト配列 (リニアメモリ) として提供しており，Wasm アプリケーションはリニアメモリ外の共有メモリに直接アクセスできない．そこで，TZmediator は POSIX API を拡張し，Wasm アプリケーションがリニアメモリだけでなく共有メモリも操作できるようにする．Wasm ランタイムにおいてアドレスの判定を行い，それぞれのメモリにアクセスする．リニアメモリのアドレスは Wasm アプリケーション内でのみ有効であるため，リニアメモリのアドレスはホストアドレスに変換してアクセスする．

Wasm アプリケーションのシグナルハンドラを実行できるようにするために，TZmediator は Wasm の間接呼び出しを利用する．Wasm では関数が関数インデックスによって識別されるため，シグナルが TA に送信された時に，Wasm ランタイムが登録された関数を直接実行することができない．そのため，Wasm ランタイムは登録されたインデックスを使用して，対応する関数を間接的に呼び出す．この間接呼び出しを可能にするために，Wasm アプリケーションはリンク時にシグナルハンドラとして使用される関数をエクスポートする．

5. 実験

TZmediator を用いてクラウドアプリケーションにおける CA と TA 間の通信性能を測定する実験を行った．測定にはプロセス間通信の性能を測定する IPC-Bench [21] を一部，改変して用いた．本実験には，ARM Cortex-A53, 2GB のメモリを搭載した Armadillo-X2 [22] を使用した．セキュアワールドでは OP-TEE 3.15.0 を実行し，ノーマルワールドでは Linux 5.10 を実行した．



(a) 通信性能　　(b) システム性能

図 7: 転送方式が性能に及ぼす影響

5.1 通信性能とシステム性能への転送方式の影響

4.3 節で述べた，TA からシャドウプロセスへの POSIX API の様々な転送方式が通信性能に及ぼす影響を調査した．そのために，IPC-Bench に含まれる名前付きパイプと TCP ソケットを用いて 4 KB のデータを送受信し，CA と TA 間の通信レイテンシを測定した．従来 RPC，軽量 RPC，ポーリングベースの方式を用い，ポーリングでは待機時間を 0 または 1 µs に設定した．この実験では TA はネイティブアプリケーションを実行した．図 7(a) に示すように，従来 RPC と比較して，軽量 RPC ではレイテンシが 84〜85%低減した．これは，ワールドを切り替える回数が削減されたためである．待機時間を設けないポーリングでは，ワールド切り替えが発生しないため，レイテンシは最も低く，49〜52 µs であった．一方，待機時間を 1 µs に増やすと，スリープ処理のためにワールド切り替えが発生するため，レイテンシは大きく増加した．このレイテンシは軽量 RPC よりも大幅に高かったものの，従来 RPC よりは低かった．

次に，TZmediator がシステム全体に与える影響を調べるために，ノーマルワールドで sysbench 1.0.20 [23] を実行し，4 スレッドで CPU ベンチマークのスコアを測定した．実行中はレイテンシの測定実験と同様に，CA と TA 間の通信を継続的に行った．図 7(b) に示すように，レイテンシとは異なり，従来 RPC が最も高いスコアを達成した．これは通信頻度が最も低かったためである．軽量 RPC では通信頻度が増加したため，システム性能は 0.2〜6.6%低下した．一方，待機時間を設けないポーリングでは，従来 RPC と比較してシステム性能が 30〜32%低下した．これは通信頻度の増加に加え，ビジーウェイトによる CPU オーバヘッドが大きくなったためである．また，ポーリング時に待機時間を挟むことでシステム性能が向上することが分かった．

ビジーウェイトによるポーリングは最も高い通信性能を達成したが，CPU オーバヘッドの影響によりシステム性能は最も低かった．そのため，この転送方式は最も性能重視である．一方，軽量 RPC は通信性能とシステム性能ともに 2 番目に優れていたことから最もバランスの取れた転送方式であると言える．そのため，以降の実験ではこれら 2 つの転送方式を使用した．

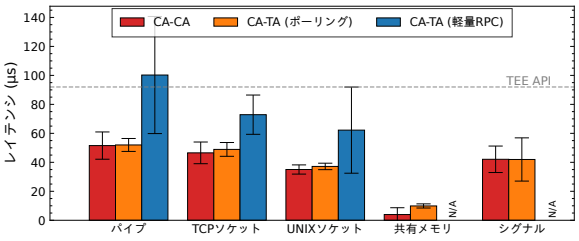


図 8: 既存の通信との通信性能の比較

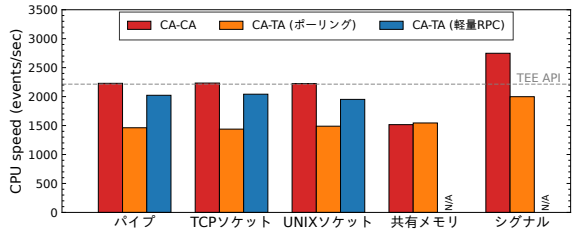


図 9: 既存の通信とのシステム性能の比較

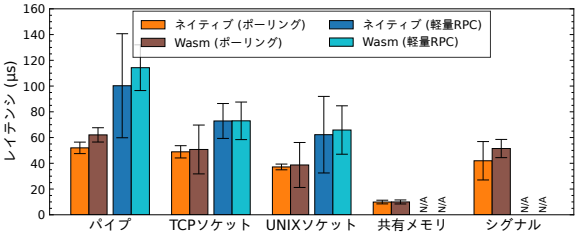


図 10: Wasm が通信性能に及ぼす影響

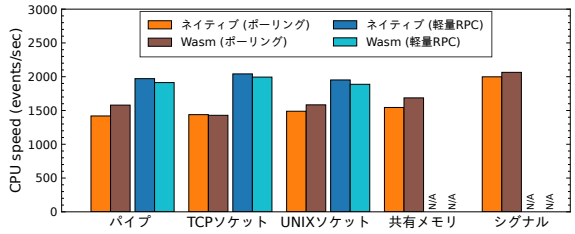


図 11: Wasm がシステム性能に及ぼす影響

5.2 既存の通信との比較

2 つの CA 間で通信する場合と，CA と TA 間で TEE API を用いる場合の通信性能を測定し，TZmediator を用いる場合と比較した．2 つの CA 間における通信では，CA は TEE Client API を使用せず，通常のプロセスとして IPC-Bench を実行する．TEE API を用いる場合，CA は TA と共有するメモリに 4 KB のデータを書き込み，TEE Client API を用いて TA のコマンドを実行する．TA は TEE Internal Core API によって呼び出されると，そのデータを読み取り，共有メモリに 4 KB のデータを書き込む．コマンドの実行終了後，CA は共有メモリからデータを読み取る．

実験結果を図 8 に示す．CA-CA 間の通信と比較して，TZmediator のレイテンシはポーリング方式において最大 5.9 µs 増加した．一方，TEE API による通信と比較すると，軽量 RPC を使用した場合は，パイプのレイテンシは 8.3 µs 増加したが，ソケットのレイテンシは 19〜30 µs 減少した．また，通信中のシステム性能について測定した結果を図 9 に示す．システム性能は CA-CA および TEE API の方が高い結果となったが，TEE API と比較して，軽量 RPC を使用した場合の性能低下は 8.6〜12%にとどまることが分かった．

5.3 Wasm の実行オーバヘッド

TA で実行される Wasm アプリケーションと CA 間の通信性能およびシステム性能を測定し，TA でネイティブアプリケーションを実行した場合と比較した．図 10 に示すように，Wasm アプリケーションのレイテンシは 0.1〜14 µs 増加することが分かった．これは Wasm と WASI のオーバヘッドによるものであるが，その増加量は限定的であり，ネイティブの実行に近い性能を維持できることが分かった．図 11 に示すように，システム性能は軽量 RPC を用いた場合，Wasm アプリケーションにより最大 3.3%低下したが，ポーリングを用いた場合は逆に最大 11%向上した．これは Wasm アプリケーションの実行オーバヘッドによりポーリング頻度が低下したためと考えられる．

5.4 スケーラビリティ

TZmediator のスケーラビリティを検証するため，実行するクラウドアプリケーションの数を 4 つまで増やしたときのレイテンシを測定した．各アプリケーションは 1 つの CA と 1 つの TA で構成され，CA と TA は名前付きパイプを使用して相互に通信した．TA はネイティブアプリケーションまたは Wasm アプリケーションのいずれかを実行した．

図 12 に平均レイテンシを示す．ポーリングを使用した場合はアプリケーション数が少ない間はレイテンシが低く抑えられるが，4 つのアプリケーションを同時に実行すると，ネイティブアプリケーションで 2.9 倍，Wasm アプリケーションで 3.9 倍にレイテンシが増加した．一方，軽量 RPC を使用した場合はアプリケーション数が増えるにつれてレイテンシが徐々に増加し，4 つのアプリケーションを実行した場合のレイテンシはポーリングよりも 28〜93 µs 低くなることが分かった．この結果から，スケーラビリティの観点では軽量 RPC の方が優れていると言える．

5.5 実アプリケーションの性能

TZmediator を用いた実アプリケーションの実行性能を検証するために，POSIX API を用いて協調実行を行うように DarkneTZ [16] を修正した．DarkneTZ はモデル分割と TrustZone を組み合わせることでディープニューラルネットワーク (DNN) の攻撃対象領域を縮小するフレーム

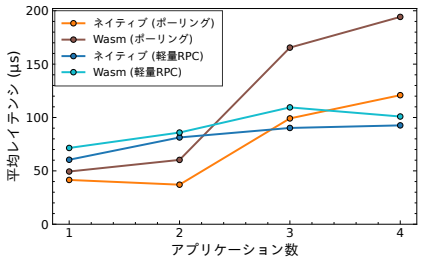


図 12: TZmediator のスケーラビリティ

ワークであり，TEE API を用いて実装されている．この実験では 9 層目のソフトマックス層をセキュアワールドで実行し，残りの 9 層をノーマルワールドで実行した．そして，手書き数字画像の MNIST データセットで事前学習済みのモデルを使用して推論を行った．POSIX 共有メモリを使用した DarkneTZ では，元の実装と比較して，推論時間が 2.9%増加することが分かった．また，名前付きパイプを使用すると，ポーリングの場合でも推論時間が 14%長くなることが分かった．

6. 関連研究

Intel SGX [7] はプロセス内部にエンクレイヴと呼ばれる保護領域を作成し，その中でアプリケーションを安全に実行することを可能にする．エンクレイヴ内で動作するアプリケーションに POSIX API を提供するシステムがいくつか提案されている．SCONE [24] はエンクレイヴ内のアプリケーション向けに専用の C ライブラリを提供する．このライブラリはエンクレイヴと外部の間でコンテキストを切り替えずに非同期システムコールを用いて OS を呼び出す．このシステムコールを利用することで，エンクレイヴアプリケーションは OS を介して他のプロセスと通信することができる．

複数のエンクレイヴ間で POSIX API を用いて協調実行を可能にするシステムも提案されている．Graphene-SGX [25] (現在の Gramine) はエンクレイヴ内のアプリケーションにライブラリ OS を提供しており，複数のエンクレイヴ間でライブラリ OS を介して直接通信することができる．また，ライブラリ OS は `fork` や `execve` をサポートしており，新たなエンクレイヴの生成や実行を可能にする．Occlum [26] もライブラリ OS を提供しており，スレッドを用いてエンクレイヴ内部で複数のアプリケーションを実行することができる．

eMCOS POSIX [27] は VOSySmonitor ハイパーバイザ [28] を用いて TrustZone のセキュアワールド上で動作する POSIX 互換性を備えたリアルタイム OS である．POSIX API を用いた複数プロセスおよびスレッド間の協調実行を可能にする．しかし，プロセス間およびスレッド間通信はセキュアワールド内部に限定されており，ノーマルワールドで動作するプロセスとの通信手段は提供されていない．さらに，リアルタイム OS ではプロセスやスレッドが高い優先度でスケジューリングされるため，クラウドアプリケーションの実行には必ずしも適していない．

WebAssembly System Interface (WASI) [18] は Wasm 向けのシステムインタフェースであり，現在標準化が進められている．WASI Preview 1 では POSIX に類似した API が提供されているものの，ソケット機能は限定的である．WASI Preview 2 ではソケット機能が大幅に拡張されている．POSIX 互換性の向上を目的とした WASI の拡張として WASIX [29] が提案されている．また，WebAssembly Linux Interface (WALI) [30] は Linux システムコールのインタフェースを提供し，Linux アプリケーションを Wasm アプリケーションとして実行することを可能にする．

ReZone [3] は TrustZone のセキュアワールドを分割して，TA を隔離して実行する仕組みを提供する．TA ごとにゾーンと呼ばれるサンドボックス環境を作成し，それぞれに独立した Trusted OS を提供する．各ゾーンは割り当てられた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は制限される．このため，TA として動作するクラウドアプリケーションが侵害された場合においてもゾーン外への攻撃を防ぐことができる．各 TA に割り当てられるリソース量が少なくなるが，TZmediator は最小限の処理のみを TA として実行するため，WebAssembly の代わりに ReZone と組み合わせて利用することも考えられる．

7. まとめ

本稿では，Arm TrustZone の 2 つのワールドに分割されたクラウドアプリケーションに対し，POSIX API を用いて協調実行を行うことを可能にする 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，より強い隔離が必要な場合にはそれらを Wasm アプリケーションとして実行する．ワールド間にまたがって POSIX API を用いた通信を実現するために，TZmediator は各 TA に対応するシャドウプロセスをノーマルワールドに作成する．TA は POSIX API の呼び出しをシャドウプロセスに転送し，シャドウプロセスが標準ライブラリを用いて TA に代わって POSIX API を実行する．実験により，クライアントアプリケーションにおける CA と TA 間の通信性能を確認した．

今後の課題は，複数の CA と TA が並列に動作し，POSIX API を介して協調する様々なクラウドアプリケーションを開発することである．このようなクラウドアプリケーションにより，より効率的な協調処理が実現できると期待される．

謝辞 本研究の一部は，JST，CREST，JPMJCR21M4 の支援を受けたものである．

参考文献

[1] International Data Corporation: IDC's Worldwide Edge Spending Guide Taxonomy, 2025: Release V1 (2025).
[2] Shi, W., Cao, J., Zhang, Q., Li, Y. and Xu, L.: Edge Computing: Vision and Challenges, *IEEE Internet of Things Journal*, Vol. 3, No. 5, pp. 637–646 (2016).
[3] Cerdeira, D., Martins, J., Santos, N. and Pinto, S.: ReZone: Disarming TrustZone with TEE Privilege Reduction, *Proc. Security Symp.*, pp. 2261–2279 (2022).
[4] GlobalPlatform: GlobalPlatform Device Technology TEE Client API Specification Version 1.0 (2010).
[5] GlobalPlatform: GlobalPlatform Technology TEE Internal Core API Specification Version 1.3.1 (2021).
[6] IEEE: IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 8 (2024).
[7] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V. and Savagaonkar, U. R.: Innovative instructions and software model for isolated execution, *Proc. Int. Workshop on Hardware and Architectural Support for Security and Privacy* (2013).
[8] ARM Ltd.: ARM Security Technology – Building a Secure System Using TrustZone Technology, White Paper (2009).
[9] Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K. and Song, D.: Keystone: An Open Framework for Architecting Trusted Execution Environments, *Proc. European Conf. Computer Systems* (2020).
[10] Advanced Micro Devices, Inc.: Secure Encrypted Virtualization API Version 0.24 (2020).
[11] Intel Corporation: Intel Trust Domain Extension, White Paper (2023).
[12] ARM Ltd.: Arm Confidential Compute Architecture, https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.
[13] Linaro Ltd.: OP-TEE, https://www.trustedfirmware.org/projects/op-tee.
[14] Open-TEE: Open-TEE, https://open-tee.github.io/.
[15] Google Inc.: Trusty TEE — Android Open Source Project, https://source.android.com/docs/security/features/trusty.
[16] Mo, F., Shamsabadi, A. S., Katevas, K., Demetriou, S., Leontiadis, I., Cavallaro, A. and Haddadi, H.: DarkneTZ: towards model privacy at the edge using trusted execution environments, *Proc. Int. Conf. Mobile Systems, Applications, and Services*, pp. 161–174 (2020).
[17] Menetrey, J., Pasin, M., Felber, P. and Schiavoni, V.: WaTZ: A Trusted WebAssembly Runtime Environment with Remote Attestation for TrustZone, *Proc. Int. Conf. Distributed Computing Systems*, pp. 1177–1189 (2022).
[18] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A. and Bastien, J.: Bringing the web up to speed with WebAssembly, *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 185–200 (2017).
[19] Bytecode Alliance: WebAssembly Micro Runtime (WAMR), https://github.com/bytecodealliance/wasm-micro-runtime.
[20] OP-TEE: Trusted Applications, https://optee.readthedocs.io/en/latest/building/trusted_applications.html.
[21] Goldsborough, P.: IPC-Bench, https://github.com/goldsborough/ipc-bench?tab=readme-ov-file.
[22] Atmark Techno, Inc.: Armadillo-X2, https://armadillo.atmark-techno.com/armadillo-x2.
[23] Kopytov, A.: sysbench, https://github.com/akopytov/sysbench/tree/1.0.20.
[24] 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, *Proc. Symp. Operating Systems Design and Implementation*, pp. 689–703 (2016).
[25] Tsai, C., Porter, D. E. and Vij, M.: Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX, *Proc. Annual Technical Conf.*, pp. 645–658 (2017).
[26] 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, *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 955–970 (2020).
[27] eSOL Co., Ltd.: eMCOS POSIX - High Performance, POSIX-Compliant Real-Time Operating System Designed for Critical Multicore, https://www.esol.com/embedded/product/emcos-posix_overview.html.
[28] Lucas, P., Chappuis, K., Boutin, B., Vetter, J. and Raho, D.: VOSYSmonitor, a TrustZone-based Hypervisor for ISO 26262 Mixed-critical System, *Proc. Conf. Open Innovations Association FRUCT*, pp. 231–238 (2018).
[29] Wasmer Inc.: WASIX - The Superset of WASI, https://wasix.org/.
[30] Ramesh, A., Huang, T., Titzer, B. L. and Rowe, A.: Empowering WebAssembly with Thin Kernel Interfaces, *Proc. European Conf. Computer Systems*, pp. 1–20 (2023).