Vol.2020-OS-150 No.7
2020/7/30

TEEによるスケーラブルかつチート耐性のあるオンラインゲームアーキテクチャ

畑輝史[1]　オブラン・ピエールルイ[1]　河野健二[1]

概要：オンラインゲームにおけるスケーラビリティ向上のため，ゲームのコア・ロジックをクライアンサイドに移譲するようになっている．クライアントサイドのコア・ロジックは原理的に覗き見・改竄が可能であり，いわゆるチート行為の防止・検出が困難となっている．本論文ではチート耐性とスケーラビリティを同時に達成するアーキテクチャを提案する．チート耐性を高めるために Trusted Execution Environment (TEE) を使用し，スケーラビリティのためにピア・ツー・ピア型を採用する．コア・ロジックを TEE 内で実行することで，悪意のあるクライアントからセンシティブなデータおよびコードを保護し，覗き見や改竄を防止する．TEE として Intel SGX を利用し，ZDoom という実際のファーストパーソン ・シューティング型のゲームに適用したところ，原理的にチート耐性が向上しているにもかかわらず，チート耐性の低いアーキテクチャと同等のスケーラビリティを達成できることを示す．

1. はじめに

オンラインゲーム産業の市場規模は 2020 年では 900 億ドルに達し, ゲーム人口は 25 億人である [12]. ゲームの種類は数多くあり, ファーストパーソンシューティング [5], RPG ゲーム [14], モバイルゲーム [13] など多岐にわたる. ゲームに対してチートを行うと金銭的利益や, 注目を集めることができる. そのため, チート行為はプレイヤに不快な思いをさせるだけでなく, ゲームの大会運営 [2] やゲーム市場に打撃を与える [3].

ゲームはゲーム内のオブジェクトの状態 (マップ, アバタ, アイテムなど), プレイヤの動作の定義 (体力の初期値が 100 で, 敵の攻撃を受けると体力が 1 減るなど) から構成される．Goodman ら [17] はチート行為を “ゲームの状態をゲームのルールに反するように更新し, 不公平に有利になること” と定義している．例えばファーストパーソンシューティングでは, 不死身になるためにプレイヤが受けたダメージを更新しないなどである.

チート行為が可能となってしまう原因はゲームがプレイヤが所有するマシンで実行されることである．プレイヤはゲームのコードやゲームのオブジェクトの情報が格納されているメモリの位置を調べて変更したり, ネットワークのメッセージの不正な変更, コピー, 再送などをすることができる．アンチチートソフトウェア [11][7] は存在するが, チート行為の開発との競争となってしまっている [6][9].

チートを防ぐために, 多くのゲームはクライアント・サーバ型のアーキテクチャを採用している．中央のサーバがプレイヤの入力を検証し, 次のゲームの状態を各プレイヤに送信する．これはチート耐性はあるがスケーラビリティは低い．人気のゲームでは同時に数十人のプレイヤが集まる [10]. 世界記録は MMORPG ゲームの EVE online での 7548 人同時接続である．しかし, この記録を達成するためにはサーバの負荷に耐える必要があり, ゲームの速度を通常の 10%に落として実行した.[4]

スケーラビリティを向上させるために, 一部のゲームはサーバ側で行っていた処理をクライアント側に移す．しかし, これはチート行為が行われる可能性が高くなる．ゲームの処理はプレイヤが完全にコントロールすることができる状態で実行されるためである．例えば Wall hack というチートでは, 他のプレイヤのアバタのマップ上の座標を完全に把握する．このチートは, ゲームのプロセスにアクセスすることで敵の座標を知ることができる.

一方, ゲームアーキテクチャとして ピア・ツー・ピア型アーキテクチャがある．プレイヤはゲームの状態を中央サーバを介さずに直接他のプレイヤと交換する．このアーキテクチャはスケーラビリティは良いが [23][25], チート耐性が低い．理由は 2 つある．第一に, チータは他のプレイヤに気づかれることなくチートを行うことができる．Wall hack の例では, 他のプレイヤがマシンのメモリにアクセスしていることに気づくことはできない．第二に, 全てのプ

[1] 慶應義塾大学
Keio Uniersity

レイヤ間でゲームの状態の一貫性を保つために, 全ての情報をプレイヤ間で交換する必要がある. そのため, チータへも重要なデータをネットワークを介して送信することになる.

チート耐性のあるピア・ツー・ピア型アーキテクチャの研究は他のプレイヤの正当性を検証する処理を追加する手法を用いている. 検証者はチートを検出するために, 全てのプレイヤの行動を記録してリプレイし, 結果を比較する [26][22]. この方法では計算処理を追加しさらに応答時間が遅くなるため, チート行為を解決していないと考えられる. その上, この方法は検証者は正しく選択されるという仮定に基づいている. 検証の処理も信頼できないプレイヤのマシン上で行われるため, この仮定は現実的ではない.

この問題を解決するために, 「お守り」という我々は新たなスケーラビリティとチート耐性を同時に満たすゲームアーキテクチャを提案する. このアーキテクチャはチートを行うための手法は様々であるが, 仮にゲームのデータ, コード, ネットワークパケット全てを保護することができればチート耐性を向上させることができるという考察から得られた手法である. スケーラビリティは中央サーバを経由することのないピア・ツー・ピア型アーキテクチャによって達成する. チート耐性は Trusted Execution Environment (TEE) を利用して達成する. TEE とは CPU の特別な実行モードであり, コードとデータに対して完全性と機密性を保証する. TEE の例として Intel SGX[21] や ARM TrustZone[16], AMD SEV[1] などがあげられる. これらは多くのゲームのデバイス (コンピュータ, ゲームコンソール, スマートフォン) などで使用することが可能である. 「お守り」はゲーム開発者がコストを抑えながら運用をすることを可能にする.

「お守り」はゲームのクライアントを 2 つのコンポーネントに分ける. 1 つ目はユーザの入力や画面出力, サウンド, ネットワークを含む信頼できないコンポーネントである. 2 つ目はゲームの状態やルールを管理する処理である. これは TEE 内で動作する. ゲームごとにそのルールやデータ構造が異なるが, 「お守り」は全てのチート行為に関わるデータ構造は TEE 内で管理するという共通の方法でチートを防ぐ.

我々は P2P ゲームの自作のプロトタイプと, ZDoom[15] フレームワークを Intel SGX を用いて実験評価をした. 結果は「お守り」はピア・ツー・ピア型と同等にスケーラブルであると同時にクライアント・サーバ型と同等のチート耐性を提供することを示している.

本論文の構成を以下に示す. 第 2 章では現在のゲームのアーキテクチャとチート耐性ついて述べる. 第 3 章では「お守り」について説明する. 第 4 章では 2 つのゲームのアーキテクチャについて述べる. 第 5 章で評価を述べる. 第 6 章で関連研究を紹介する. 第 7 章で結論を述べる.

2. 現在のゲームのアーキテクチャおよびチート耐性

2.1 オンラインゲームのアーキテクチャ

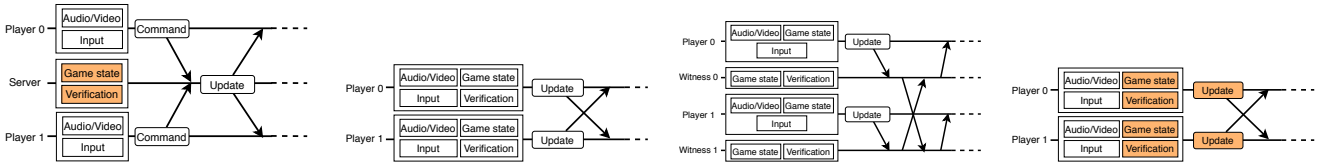
オンラインゲームはクライアント・サーバ型からピア・ツー・ピア型まで様々なアーキテクチャを採用している. クライアント・サーバ型では中央サーバがゲームの状態を管理し, 他のプレイヤから入力を集めて検証してからゲームの状態を更新する. ピア・ツー・ピア型ではそれぞれのプレイヤがクライアントとサーバの両方の役割を持ち, 中央サーバを必要としない. 両方のアーキテクチャを図 1 が示している.

2 つのアーキテクチャの間にはチート耐性とスケーラビリティ, コストのトレードオフが良く知られている. ピア・ツー・ピア型アーキテクチャと比較して, クライアント・サーバ型は高いチート耐性を達成し, 開発者のメンテナンスのが簡単になるが, コストが高くスケーラビリティが低い. 近年のゲーム産業は高スケーラビリティで高フォールトトレラントなアーキテクチャに移行している. 運用コストを減らしてスケーラビリティを向上させるために, ゲーム開発者はサーバにある機能をクライアントに移すことが求められている. 中央サーバの負荷が高いとプレイヤはサーバに接続することが難しくなる. サーバがアップデート中である場合やバグの修正, サーバがクラッシュしている最中も同様である. しかし, サーバの機能をクライアント側に移すとチート耐性が低下し, プレイヤがチートを行うための方法が解析することが可能になりゲームの状態にアクセスすることが可能になる.

2.2 オンラインゲーム上でのチート行為

表 1 はオンラインゲームでのチート行為の一覧である. チートの原因はゲームのデータやコード, ネットワークパケットへの不正のアクセスや書き換えである. チートは 3 種類に分類することができる. 1 つ目はネットワークでのチートである. これはネットワークパケットに対して操作を行うチートである. 2 つ目は書き換えである. これはゲームのコードやデータを不正に書き換えることである. ピア・ツー・ピア型は特にこのチートに対して耐性が低い. 3 つ目は不正な読み込みアクセスである. ゲームのコードやデータ, ネットワークメッセージを読むことでゲームのルール上知ることのできない情報を知るというチートである.

ピア・ツー・ピア型のアーキテクチャではチートが行われやすい. これはプレイヤが自身のアバタやゲームの状態を信頼できないマシン上で管理し, 更新情報を自分で他のプレイヤに送信するためである. 一方, クライアント・サーバ型ではチートが難しい. 信頼できる中央サーバが全ての更新情報を受け取り, 検証する. そして, ゲームの情報の露出を防ぐためにその更新情報が必要なプレイヤにのみ送信



(a) クライアント・サーバ　(b) ピア・ツー・ピア　(c) ピア・ツー・ピア + 検証　(d) 「お守り」

図 1: オンラインゲームアーキテクチャ

チート名	説明	C/S	P2P	P2P+audit	「お守り」
Escaping	不利な状況を避けるために, 意図的にコネクションを切断	–	–	–	–
Time cheat	相手の行動を見てから自分の行動を決定	–	–	–	√
Network flooding	他のプレイヤの処理を妨げるために DoS 攻撃を実行	√	√	√	√
Fast rate	ゲームのイベント周期を早める	√	–	√	√
Dead reckoning	意図的にパケットを落とし, 画面上の自分の位置をずらす	√	–	√	√
Replay	他のプレイヤのコマンドを再送信	√	–	√	√
Consistency cheat	プレイヤごとに違う更新情報を送信	√	–	√	√
Spoofing	メッセージを偽造することで他のプレイヤになりすます	√	√	√	√
Blind opponent	意図的にパケットを落とし, 自分の情報を隠す	√	–	√	√
Client-side tampering	ゲームのコードやネットワークメッセージの書き換え	√	–	√	√
Game version tampering	最新でないバージョンを使用する	–	–	–	√
Aimbots	プログラムを使い自身の行動を自動化する	–	–	–	–
Duping	アイテムなどを不正に生成する	√	–	√	√
Sniffing	ネットワーク上で交換されているメッセージを覗く	–	–	–	√
Maphack	自分の可視範囲外のオブジェクトなどの情報を得る	√	–	√	√
Rate analysis	パケットの統計情報を分析することで情報を得る	–	–	–	√
Total prevented		10/16	2/16	10/16	14/16

表 1: クライアント・サーバ (C/S), ピア・ツー・ピア (P2P), 検証つきピア・ツー・ピア (P2P-audit), 「お守り」でのチートの種類と防止可能性

する.

いくつかのチート対策ツールでは, プレイヤのメモリや実行中のプロセスの確認やチートをするプロセスの存在の確認を行っている. しかし, プレイヤのマシン上で実行されるため, ゲームのパフォーマンスを下げる原因になる. また, チータはそのチェックをすり抜けることも可能である.

エイムボットの検出は良く研究されているがアーキテクチャによらず難しい. これはエイムの精度の良いプレイヤと自動でエイムを合わせているプログラムを見分けることが難しいからである. 高度なエイムボットはメモリの中の状態とビデオバッファの両方を見ることで人間の動きを模倣する. さらに高度なチートツールは画像処理や機械学習を利用して人がプレイするのと同じようにゲーム画面から次のコマンドを入力する. 同様にエスケープチートもプレイヤが意図したのか, ネットワークのスイッチの不具合などの意図していない場合なのかの検出が難しい.

ピア・ツー・ピア型ではスケーラビリティとフォルトトレランスは高いが, チート耐性が低い. これはゲームの状態がプレイヤによって管理されているからである. プレイヤはゲームを自身のマシン上で実行するため, ソフトウェアを全てコントロールすることができる. プレイヤによってプログラムコードやデータ, ネットワークのパケットを自由に書き換えられる可能性がある.

チート耐性をもつピア・ツー・ピア型アーキテクチャはこれまで研究されてきた. これらのシステムでは他のプレイヤの行動を検証するためのソフトウェアを実行する. これについては第 6 章で詳しく説明する. このチェックは全メッセージのログの保存や, ゲームの処理のリプレイを必要とする. これは無視できない計算コストを生じる. さらにチートの影響とプレイヤ同士の共謀を制限するために, 確率的な仮定をしている.

2.3 Trusted Execution Environment and Intel SGX

Trusted Execution Environment (TEE) はホストのマシンが信頼できない場合でもプロセスをセキュアに実行するための保護機構である. TEE はソフトウェアとハードウェア両方が攻撃者によってコントロールされている場

合でもコードとデータに対して完全性と機密性を保証する．TEE の例として主に, Intel SGX や ARM TrustZone, AMD SEV などがあげられる．本研究では我々は Intel SGX を使用する．Steam の調査 [8] によると, 80%の PC ゲームマシンは Intel のプロセッサを使用しているからである．けれども, 我々のアーキテクチャは他の TEE に適用することも可能である．

2.4 脅威モデル

我々はオンラインゲームで共謀することもできるチータが複数存在することを考える．チータの目的はゲーム上で有利になることである．プレイヤはハードウェアとソフトウェアの両方を全てコントロールすることができる．ただし, CPU は例外で TEE を含め正しく実装されているとする．プレイヤは絶対に TEE 内のコードやデータにアクセスすることができない．TEE に対するサイドチャネルアタックは脅威モデルの対象外とする．同様に我々は可用性も対象外とする．これはチータが他のプレイヤとの通信を止めることができるからである．また, ゲームのコードは既に知られていて, チータから研究されていると仮定する．

3. お守り

「お守り」はスケーラビリティとチート耐性を同時に達成するゲームアーキテクチャである．(i) スケーラビリティのために, ピア・ツー・ピア型を採用する．(ii) ゲームの状態を保護するために TEE を使用する．(iii) 他のプレイヤとのコミュニケーションを全て暗号化する．「お守り」のアーキテクチャの概略は図 1d の通りである．

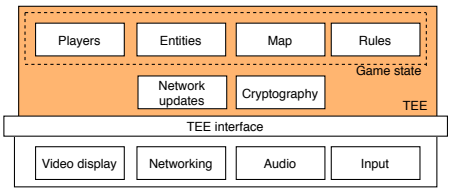


図 2: 「お守り」のアーキテクチャ

本章の構成を以下に示す．3.1 節ではゲームの状態をセキュアにすることについて述べる．3.2 節ではコミュニケーションについて説明する．3.3 節では「お守り」でチートを防ぐ方法について述べる．**??**節ではゲームのプロセスのライフサイクルにいて説明する．

3.1 セキュアなゲームの状態

ゲームの状態を保護するため, 「お守り」はデータとコードをチータからアクセスできないように TEE に移す．TEE にゲームの状態を移動させるとチータからコードとデータの書き換えや読み込みを防ぐことができる．これにより, チータはチート行為に必要なコードやデータにアクセスすることができないのでチートを行うことができない．

単純な方法としてゲーム全体を TEE にいれるという方法が考えられる．しかし, この方法では問題がいくつか生じる．(i) コードの多くを TEE 内に置くことは Trusted Computing Base が増え, TEE 内のコードのバグの量が増える原因になる．(ii) I/O を TEE 内から行うことは難しく, 信頼できないコードを一部 TEE の外に置く必要がある．(iii) ゲームはスクリプトエンジンやデバッグコンソールが備わっており, うまく切り離さなければチータから簡単にチートを行われてしまう．

全てのゲームはそれぞれ違うルール, 違うデータ構造を持っている．その上で, 我々はどのデータやコードを TEE にいれるべきかの指標を与える．チートを行う上で必要なデータは全て TEE 内で保護されなければならない．例えば, 場所の不正な移動, 他のプレイヤの場所の取得, 自身の体力や装備の変更などのチートに必要なプレイヤのデータ構造やゲームマップの情報が該当する．

3.2 セキュアなコミュニケーション

センシティブなコードやデータを TEE に入れるだけではチートを防ぐことに関して不十分である．チータはネットワークメッセージを読むことでセンシティブな情報を学習することができ, メッセージを書き換えや作成により有利になることができる．

このようなチートを防ぐために, 「お守り」は (i) 全てのネットワークのメッセージの暗号化, (ii) 完全性の保護, (iii) パーシステントカウンタの利用, (iv) 定期的な更新情報の送信を行う．メッセージの作成, 暗号化, 復号化は TEE 内で行われる．チータは暗号化の鍵やメッセージの平文にアクセスすることができない．完全性の検証も行うため, メッセージを書き換えたかどうかは検証することが可能である．

ロールバック攻撃を防ぐために, 「お守り」はパーシステントカウンタを使用する．カウンタの一貫性とフォールトトレランスを守るため, ROTE protocol[24] を使用する．メッセージはそれぞれカウンタを持ち, 他のプレイヤが検証することができる．

最後に, 「お守り」は必ず定期的に更新情報の交換を行う．ゲームの状態に変更がなく, 更新情報がない場合は代わりに空の更新情報を送信する．メッセージの暗号化によって TEE の外側からはメッセージが空であるということはわからない．定期的なメッセージの更新によってチータがネットワークの頻度などの統計情報からゲームの状態を学習することを防ぐ．

3.3 チート行為

表 1 は「お守り」が防ぐことのできるチートの種類を示している．「お守り」はクライアント・サーバやピア・

ツー・ピア+検証より 4 種類, ピア・ツー・ピアより 12 種類多くのチートを防ぐことができる. これはチートをするために必要な重要なコードやデータにチータがアクセスできないためである. ネットワークメッセージも暗号化されているためチータが平文を読むことはできない. さらに, ネットワークメッセージの MAC によりメッセージの書き換えも検出可能である.

他のアーキテクチャと比較すると「お守り」は *time cheat*, *sniffing*, *rate analysis*, *version tampering* を防ぐことができる. 「お守り」によりチータはコードのコントロールフローを変えることができないため time cheat を行うことはできない. ネットワークメッセージの暗号化により sniffing は防ぐことが可能である. 他のアーキテクチャでも暗号化は可能であるが, チータは暗号化の鍵にアクセスすることができる. しかし, 「お守り」は暗号化の鍵を enclave の中に保存するためチータはアクセスすることができない. Rate analysis はネットワークメッセージの大きさと送受信の頻度を一定にすることで防ぐことができる. Version tampering は ROTE protocol により防ぐことができる.

他のアーキテクチャ同様に防ぐことができないチートも存在する. Aimbots はゲームの上級者と自動化されたプログラムの違いを判断することは難しいため, このチートを防ぐことはできない. 「お守り」は Aimbots の検出率を向上させることはできるが, 100% 検出することは不可能である. Escaping に関して「お守り」はプレイヤが enclave を正常終了したかどうかを検出することは可能である. しかし, 外的要因による接続の切断かどうかは判断することができない.

4. 2 つのゲームのアーキテクチャ

本論文では 2 つのゲームについて考える. 1 つ目はシンプルなゲームのプロトタイプ, 2 つ目は Doom のゲームの ZDoom[15] と呼ばれるフレームワークである. この章では我々は内部のデータ構造やネットワークのコミュニケーションの方法について述べる. そして, どのように「お守り」に組み込むのかを説明する.

4.1 ゲームのプロトタイプ

我々が作成したプロトタイプはゲームのコアロジックにのみに集中し, グラフィックインタフェースを必要としない. これにより, 評価や結論をまとめやすくした. 我々のプロトタイプはターンベースリアルタイム戦略ゲームを模している. それぞれアバターで表される複数のプレイヤーがチェスと同様にセルで構成される長方形のマップを移動し, 互いに攻撃をする. それぞれのプレイヤは自分の周りのマス (可視範囲) のみを見ることができる. それ以外のマスを見ることはできない.

ゲームは次のように進行する. プレイヤは連続するゲームフレームで様々なコマンドを実行する. 1 つのフレームはプレイヤの現在の入力, 入力を他プレイヤーへ送信, 他プレイヤの入力の受信, ゲームの状態の更新で構成される. 別のセルに移動すると可視範囲が変化する. 他のプレイヤへ攻撃を加えて体力を減らすことができる.

1) 状態*:* ゲームの状態は (i) プレイヤ (ID やニックネームを含む), (ii) ゲームマップ (セルの配列), (iii) アバタ (位置するマップの座標, コントロールしているプレイヤの ID, 体力, 移動速度), (iv) ゲームのフレームの番号の 4 つで構成される.

2) 行動*:* プレイヤは他のセルに移動し, 隣のプレイヤに対して攻撃を行うことができる. 1 つのフレームでプレイヤが行うことができる行動の数は限られている. 攻撃をされたプレイヤの体力は減少する. プレイヤは体力が 0 となったら敗退である. プレイヤは他の敵が可視範囲に入った場合のその敵を見ることができる. 可視範囲外にいる敵は知ることができない.

3) ネットワークコミュニケーション*:* 我々のゲームのプロトタイプではネットワークコミュニケーションとして TCP を使用した. 1 フレーム間に実行したいコマンドを含む 13-byte のパケットを送信する. そして, 他のプレイヤからのコマンドを受け取り, 最終的にゲームの状態を更新する. コミュニケーションは同期されており, それぞれのプレイヤは他の全てのプレイヤからのメッセージの受信を終えるまで待機する. この方法はターンベースゲームではよく採用される.

4) チート行為*:* チータは様々な方法でゲームのルールを破ることができる. 例えば, ID を偽造して他のプレイヤのなりすまし, 可視範囲の変更, 離れたセルへの移動, 攻撃されることの先読み, 自身の体力を更新しないなどを行うことができる.

4.2 ZDoom

ZDoom は C++ で書かれたオープンソースのファーストパーソンシューティングゲームである. 我々が ZDoom を選択した理由はピア・ツー・ピア型ネットワークコミュニーケーションをサポートし, オープンソースであり, メンテナンスが良いためである.

ZDoom はレベルと呼ばれるデータのセットで構成される. それぞれのレベルではプレイヤが 3D のマップを移動し, コンピュータによって操作されている敵を倒す. そして, 倒した敵の数をプレイヤ同士で競い合う. プレイヤ自身が倒されると既定の場所で復活する. ZDoom はゲームのペースが速いので, ネットワークは低遅延で多くの更新を送受信する必要がある.

1) 状態*:* 簡単のために, グラフィクスやサウンドに関連するデータ構造は省略する. それぞれのレベルは *FLevelLocals* で構成され, メタデータ (レベルの名前など), ゲー

ムの設定 (重力, 敵の数など), ゲームのルール (ジャンプなどの動作の有無), マップの形状などを含む. それぞれのプレイヤや敵は *AActor* というオブジェクトで表され, マップ上の位置や体力, 所持している武器や防具についての情報を持っている. プレイヤはそれ以外に *player_t* というオブジェクトも持ち, 倒した敵の数などの記録を所持している. 全ての *AActor* は *DThinker* を継承する. *DThinker* は *tick()* メソッドというアップデートを行うための関数をもち, マップ上のオブジェクトの更新が必要な時に呼ばれる. プレイヤの位置や体力などを変更する.

2) 行動*:* プレイヤと敵の動きは同じであり, マップ上の移動, 銃の発砲である. さらにプレイヤはマップ上のオブジェクトの拾得, ドアの開閉, 使用する武器の種類の切替なども行うことができる. マップ上のオブジェクトはプレイヤによって拾われると, 一定時間後に同じ場所に復活する. プレイヤは倒されると *SelectRandomDeathmatchSpot()* 関数で選択された場所に復活する.

3) ネットワークコミュニケーション*:* プレイヤはユーザの入力 (マウス, キーボード入力など) を交換する. ゲームの状態を直接交換するのではない. プレイヤはそれぞれ, 他のプレイヤからの入力を集めマップ上の全てのプレイヤの動きを計算する必要がある. そのため, 1 つのアップデートでも逃してしまうとゲームの状態の一貫性が保てなくなる. 一貫性が保たれていない状態を解消するプロトコルは存在しないので, その場合は一度セッションを退出して再接続する必要がある. パケットには, パケットの識別子, フレーム番号, ユーザの入力, 完全性の検証のためのアバターの位置と体力から計算するハッシュ値などが含まれる.

4) チート*:* 上記の通り, プレイヤは入力とゲームの状態のハッシュを交換するため, チートを行うことが難しい. チートをするために不正にアバタのいちを変更した場合は一貫性が保たれていない状態となり, ゲームセッションの切断, 再接続が必要となる. けれども, チートは可能である. 特に, time cheat や consistency, sniffing, map hack のチートを行うことができる.

4.3 実装

我々の実装ではネットワークパケットの暗号化に AES-GCM アルゴリズム (SGX SDK ライブラリが提供) を使用した. このアルゴリズムは機密性と完全性の両方を保証する. ネットワークのパケットはベースラインの実装より 32byte しか増加しない. そのうち 16byte は MAC に使われ, 8byte はパーシステントカウンタ, 8byte はゲームのバージョンである.

1) コード行数*:* 我々のプロトタイプは合計で 2006 LoC である. そのうち enclave は 873 LoC で, 全体の 44%である. 我々のゲームのプロトタイプは最小限の機能で構成されているのにもかかわらず, 半分以上のコードは保護される必要がないということは興味深いと考えられる. 一方, ZDoom は合計 590,000 LoC で構成され, そのうち enclave に入れたのは 35,900 LoC で全体の 6% である.

2) Enclave インタフェース*:* Enclave のインタフェースの数は少なく, それぞれ 3 つと 5 つの ecall である. それぞれ 1 つの ecall は enclave の初期化に使用し, それ以外はネットワークのメッセージの作成や操作, ゲームの更新を行う.

3) Enclave の状態*:* どちらのゲームでもプレイヤ, アバタ, ゲームマップ, ゲームオブジェクト, これらを更新するコードを enclave にいれる.

これは ZDoom では難しい課題である. 第一にゲーム中に発生するイベントは専用のスクリプトで書かれ, バイナリに変換される. 実行時に VM がスクリプトを実行する. そのため, 我々は VM が enclave 内にある様にシュミレートする ecall を作成した. 第二に ZDoom はグローバル変数を数多く使用している. どのグローバル変数を enclave に入れるべきかどうかを判断し, 依存関係を解決することはとても複雑な作業となる. そのため, 我々は enclave 内外の両方にグローバル変数のコピーを配置した. これらのコピーは同期される必要があり, enclave 内のセンシティブな情報を外に晒すことになる. この仕組みにより enclave の設計を簡略化する. 現時点では実装は途中である.

これらの 2 つの課題はソフトウェアの設計に由来し,「お守り」の設計によって生ずるものではない.

5. 評価

我々は第 4 章で紹介したゲームのプロトタイプと ZDoom に対して実験評価を行った.

我々の評価は (i)「お守り」は様々なチート行為に対して有効である, (ii)Intel SGX のオーバーヘッドとネットワークパケットの暗号化は非常に低い, (iii)「お守り」はスケーラブルである, (iv)「お守り」は実際のマルチプレイヤゲームセッションでもプレイヤの操作性に影響しない, ことを示す.

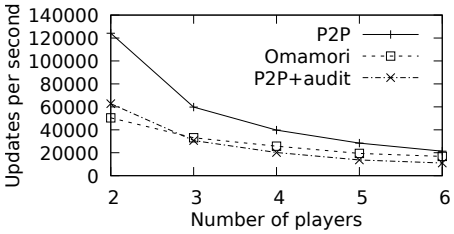
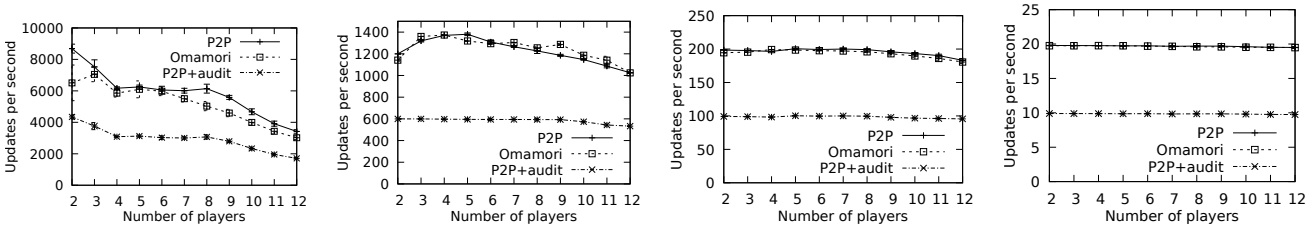


図 3: ゲームプロトタイプのパフォーマンス

5.1 実験環境

本実験では 2 台のマシンを使用した. 各マシンには Intel Core i5-8500 CPU@3Ghz(6 cores), 16GB RAM, 1Gbps の



(a) No delay. (b) 1ms delay. (c) 10ms delay. (d) 100ms delay.

図 4: ネットワーク遅延の秒間アップデート数への影響

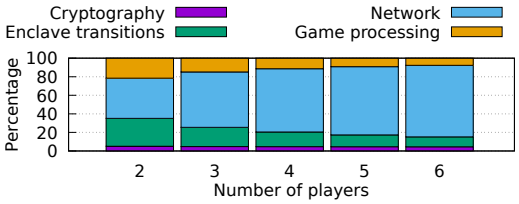


図 5: 「お守り」の処理時間の内訳

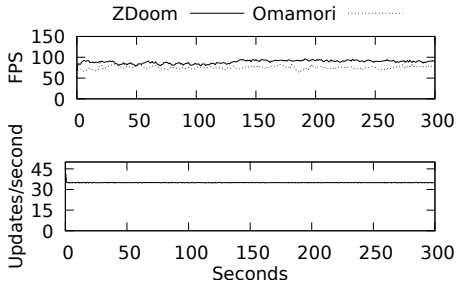


図 6: ZDoom のゲームプレイ

ネットワークが備わっている. これらのマシン上で, Ubuntu 18.04, Intel SGXSDK v2.5 を使用した. 実験は標準偏差が非常に小さくなるまで複数回行う. 1 つのコア上では最大で 1 プレイヤが実行される.

5.2 Intel SGX によるオーバーヘッド

この章ではゲームのプロトタイプを 3 つ設定で比較を行った. (i) P2P 型, (ii) P2P 型+「お守り」, (iii) P2P 型+検証. この実験は 1 つのマシンで最大で 6 プレイヤを実行した.

結果は図 3 に示した通りである. P2P 型アーキテクチャと比較して, P2P+検証はプレイヤの人数によらず 50%程度のオーバーヘッドがあった. これは検証のためのネットワークコミュニケーションによるものである. これに対し, 「お守り」のオーバーヘッドは 2 プレイヤで 59%, 6 プレイヤで 21%である. アップデートの数はプレイヤの数が増えるにつれて減少する. これはゲームの更新を全てのプレイヤに送信するためである.

次に「お守り」を使用したゲームのプロトタイプの処理時間の内訳を計測した. 図 5 からわかる通り, 暗号化復号化の処理は全体の 5%である. enclave 内外の移動のコストは 2 プレイヤの場合は 30%, 6 プレイヤの場合は 10%である. プレイヤの人数が増加するに従い, このコストの割合は減少する. 最もコストの高い処理はネットワークのコミュニケーションであり, プレイヤの人数が増加するに従いこの割合も増加する. よって, P2P 型と比較した「お守り」のオーバーヘッドはプレイヤの人数が増加するにつれて減少する.

5.3 ネットワーク遅延の影響

本実験ではネットワークの遅延がどの程度「お守り」のオーバーヘッドに影響するのかを計測した. 最大で 12 プレイヤを 2 台のマシンに振り分けて, マシン間のネットワークの遅延を Linux tc コマンドで擬似的に発生させた. ネットワーク遅延は 1ms, 10ms, 100ms に設定して実験を行った.

結果は図 4 に示した通りである. ネットワークの遅延を増加させるとアップデートの数は減少する. 12 プレイヤの「お守り」では遅延なしの場合は 3420 回であるが, 100ms の場合は 19.5 回となる. 遅延が増加すると「お守り」を使用することによるオーバーヘッドは減少する. 「お守り」のオーバーヘッドの最大値は 2 プレイヤで遅延なしの場合で 25%であるが, 遅延が 100ms の場合ではほとんどない. 前の実験で示したのと同様に, 検証つき P2P 型アーキテクチャは 50%のオーバーヘッドを生じる.

5.4 ZDoom

ZDoom を Intel SGX の enclave に移すことは非常に難しい. この実装はまだ途中である: (i) パケットの生成, 暗号化, 復号化は enclave 内で行われる. (ii) ネットワークパケットを元にゲームの状態を更新する処理の場所に ecall を配置した. けれども, 「お守り」を完全に実装した場合に近いパフォーマンスとなる.

この実験では実際に 11 人でゲームを行い, 実行中の秒間のフレーム数と更新の数を記録した. 「お守り」を使用しない ZDoom と使用する ZDoom の 2 つを実験した.

図 6 に示す様に, 「お守り」を使用した場合としてない場合の両方とも 5 分間同じ秒間フレーム数であった. 差は最大で 14%であり, 平均的には 89FPS から 76FPS 程度である.

SGX のスレッドの開始と終了時に特別な enclave の移

動の処理が実行される．このオーバーヘッドは複数のプレイヤを 1 つのマシンで動作させている際に発生する．さらに, 秒間の更新数は両方の場合で一定であり, 35 回程度であった．

6. 関連研究

2.2 で述べた通り, P2P ゲームでチート耐性を向上させるこれまでの研究は検証のための処理を追加する．

Watchmen[26] は各プレイヤに対してプロキシノードを追加する．このプロキシはメッセージをプレイヤが送受信するフィルタリングし, メッセージからチートの可能性を検証することによりプレイヤに見せるデータを最小限に制限する．プロキシはプレイヤと同じマシン上で実行されるため, 同様に改竄される可能性ある．プロキシの改竄の影響を抑えるために, プロキシはランダムに変更される．PeerReview[20] と FullReview[19] ではプレイヤが指定通りの挙動をする様にお互いに検証する．そして, 改竄防止ログを交換してこれまでのコミュニケーションをリプレイして結果を比較することで検証する．この手法は一般的な手法であり, オンラインゲームにも適用することが可能である．一方,「お守り」は追加の検証や検証のためのノードを必要としない．「お守り」はゲームの状態を TEE を利用することで保護する．

Lambda Containers[22] はモバイルアプリケーションの改竄を防止する手法を提案する．ゲーム開発者はクラウド上で各クライアントごとにコンテナを作成し, コンテナ内でクライアントをエミュレートすることで不正行為を検知する．この方法はクライアントサーバアーキテクチャを使用する．これにより,「お守り」と比較するとスケーラビリティが悪く, ゲーム開発者の負担が大きい．

Bauman らは [18] はプレイヤのコードを Intel SGX の enclave 内で実行することを提案する．「お守り」とは違い, クライアントサーバ型のゲームのコードの完全性と機密性を保証する．しかし, チートを防ぐことを目的としていない．

7. まとめ

本論文では TEE を用いることでオンラインゲームが高チート耐性と高スケーラビリティを同時に達成することを示す．我々は「お守り」という高チート耐性と高スケーラビリティをもつアーキテクチャを提案する．オンラインゲームのプロトタイプを SGX を用いて作成する．このプロトタイプによる実験では我々の提案が現実的であることを示す．

謝辞 本研究は JST, CREST, JPMJCR19F3 の支援を受けたものである．

参考文献

[1] : AMD SEV, https://developer.amd.com/sev/.
[2] : Cheating in E-sports, https://esportsobserver.com/cheating-in-esports/.
[3] : Cheating is becoming a big problem in online gaming, https://tinyurl.com/y78fg3yh.
[4] : EVE Online: the bloodbath of B-R5RB, https://www.eveonline.com/article/the-bloodbath-of-b-r5rb.
[5] : Fortnite, https://www.epicgames.com/fortnite.
[6] : I want Cheats, https://www.iwantcheats.net.
[7] : nprotect, https://nprotect.com.
[8] : Steam Hardware & Software Survey: December 2019, https://store.steampowered.com/hwsurvey/processormfg/.
[9] : System cheats, https://www.systemcheats.net.
[10] : Understanding Player unknown's Battlegrounds, https://tinyurl.com/yavtql47.
[11] : Valve Anti-Cheat System, https://support.steampowered.com/kb/7849-RADZ-6869/.
[12] : Video game statistics, https://www.wepc.com/news/video-game-statistics/.
[13] : World of Tanks, https://wotblitz.com.
[14] : World of Warcraft, https://worldofwarcraft.com/en-us/.
[15] : ZDoom, https://zdoom.org.
[16] Alves, T. and Felton, D.: Trustzone: Integrated Hardware and Software Security (2004).
[17] Baughman, N. E. and Levine, B. N.: Cheat-proof playout for centralized and distributed online games, *IEEE INFOCOM 2001*.
[18] Bauman, E. and Lin, Z.: A Case for Protecting Computer Games With SGX, *Proceedings of SysTEX '16*, pp. 4:1–4:6 (online), DOI: 10.1145/3007788.3007792.
[19] Diarra, A., Mokhtar, S. B., Aublin, P.-L. and Quéma, V.: Fullreview: Practical accountability in presence of selfish nodes, *IEEE SRDS '14*.
[20] Haeberlen, A., Kuznetsov, P. and Druschel, P.: PeerReview: Practical Accountability for Distributed Systems, *SOSP '07*.
[21] Intel: Software Guard Extensions Programming Reference, Ref. 329298-002US, https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf (2014).
[22] Kurabayashi, S.: Lambda Containers: A Comprehensive Anti-Tamper Framework for Games by Simulating Client Behavior in a Cloud, *IEEE CLOUD*, pp. 598–605 (online), DOI: 10.1109/CLOUD.2018.00083 (2018).
[23] Lehn, M., Leng, C., Rehner, R., Triebel, T. and Buchmann, A.: An online gaming testbed for peer-to-peer architectures, *ACM SIGCOMM Computer Communication Review*, Vol. 41, No. 4, pp. 474–475 (2011).
[24] Matetic, S., Ahmed, M., Kostiainen, K., Dhar, A., Sommer, D., Gervais, A., Juels, A. and Capkun, S.: ROTE: Rollback Protection for Trusted Execution, *USENIX Security*, USENIX Association (2017).
[25] Merabti, M. and El Rhalibi, A.: Peer-to-peer architecture and protocol for a massively multiplayer online game, *IEEE Global Telecommunications Conference Workshops.*, IEEE, pp. 519–528 (2004).
[26] Yahyavi, A., Huguenin, K., Gascon-Samson, J., Kienzle, J. and Kemme, B.: Watchmen: Scalable Cheat-Resistant Support for Distributed Multi-player Online Games, *ICDCS 2013*, pp. 134–144 (2013).