

# Trusted Execution Environment による オンラインゲームロジックの完全性の保護

畑 輝史<sup>1,a)</sup> オブラン ピエールルイ<sup>2,b)</sup> 河野 健二<sup>1,c)</sup>

**概要:** 近年オンラインゲームはインタラクティブ性が高まっており、ゲームの処理の負荷が高くなっている。クライアント側のハードウェアの性能の向上に伴い、サーバ側はコストを削減するためにコア・ロジックをクライアント側に移譲するのが一般的になっている。そのため、クライアント側のコア・ロジックは原理的に改竄可能となっており、いわゆるチート行為を組み込むことが容易になっている。本論文では、クライアント側のコア・ロジックの改竄を防止し、コア・ロジックの完全性を保証する仕組みを提案する。Trusted Execution Environment(TEE) 内でコア・ロジックを実行し、ゲームの状態を表すデータの完全性を TEE が担保できるようにする。これにより、コア・ロジックおよびゲームの状態の改竄が難しくなる。TEE として Intel SGX を利用し、SuperTuxKart というレーシング型のゲームに適用した結果を報告する。

## 1. はじめに

オンラインゲームの市場規模は 2020 年には 90 億ドルに達し、全世界のゲーム人口は 25 億人である。近年のゲームは非常にインタラクティブになっており、ゲームの処理負荷が非常に高くなっている。例えば、99%以上の FPS のプロゲーマーはフレームレートを 144 以上に設定している [7]。Nvidia はバトルロワイヤル系のゲームではフレームレートがプレイヤーの強さと強い相関があるため 144 以上でプレイすることを推奨している [7]。ゲームを快適に遊ぶためにはクライアント側のマシンには高機能な CPU や GPU を搭載することが必須である。インタラクティブなゲームはサーバ側の負担も非常に大きくなる。ゲームのコア・ロジックをサーバ側のみで実行するとインタラクティブな処理をするために負荷が高くなる。それに加え、サーバ側はクライアントに高頻度でネットワーク通信をする必要があるため帯域が高くなる。サーバ側の負担はクライアントの数が増えれば増えるほどさらに高くなる。近年はクライアント側のマシンの性能が向上していることもあり、ゲーム会社はサーバの負担を減らすためにサーバ側の処理をクライアントにオフロードすることが一般的になっている [10]。

しかし、クライアント側ではオフロードされた処理やデータを容易に改竄することが可能である。これにより、プレイヤーが不正な挙動を行うことができってしまうため、いわゆるチート行為が可能となる。実際に PUBG [3] という Windows ベースのオンライン対戦ゲームではプログラムやプロセスに対して DLL injection, Kernel driver attack によってチート行為が行われている [2][9]。これに対応するため、開発者たちは DLL injection や Kernel driver attack を防ぐことに注力している [2]。また、サーバ側でクライアント側の動作が不自然かどうかの検証を行なっている [9]。これではサーバ側の負荷を十分下げることができていない。

我々はオフロードされた処理を Trusted Execution Environment (TEE) 内で実行し、メモリのハッシュ値を TEE 内で管理することで完全性を達成する。オフロードされたゲームの処理を TEE 内で実行することにより、サーバ側からクライアント側で動作しているコードの検証を可能にする。ネットワークパケットを TEE 内で生成し、メッセージ認証コードを使用する。メモリ上のゲームの状態に関するデータはそのハッシュ値を TEE 内で管理する。これによって、チータがメモリやネットワークパケットを不正に改竄してもすぐに検出することが可能となる。

TEE として Intel SGX [8] を選択し、提案手法を SuperTuxKart [5] というオープンソースのゲームに適用する。適用した結果を報告する。

<sup>1</sup> 慶應義塾大学

Keio University, Japan

<sup>2</sup> 株式会社 IIJ インノベーションインスティテュート

a) terufumihata@sslab.ics.keio.ac.jp

b) pierrelouis@ijj-ii.co.jp

c) kono@sslab.ics.keio.ac.jp

## 2. 背景

### 2.1 複数人対戦型オンラインゲームのアーキテクチャ

近年のオンラインゲームのアーキテクチャはクライアントサーバ型である。複数のプレイヤーが1つのサーバに接続し、各プレイヤーはサーバのみとコミュニケーションをとる。サーバはゲームの会社によって保守管理が行われ、クライアントは各プレイヤーのマシンである。

ゲームの状態というのは各プレイヤーの操作するアバタやゲーム内の世界の地形、ルールなどを表す。これらは全てメモリ上に載っている。それぞれのアバタはゲーム内の世界での座標や自身の大きさ、重さ、形などの物理的なデータや、アバタの動作の最高速度や体力、持っているアイテム、武器の種類などのルールに関与するデータ、3Dモデルのデータなどを持っており、これらは全てメモリ上に載っている。

各プレイヤーは画面描画や音声出力のために必要な全てのゲームの状態を保持している。全てのプレイヤー間でゲームの状態が同期されることで、複数のプレイヤーが同じ世界でゲームを行うことが実現される。

各プレイヤーはキーボードなどでアバタに対して操作を行う全ての全てのプレイヤーの持つゲームの状態を更新する必要がある。例えば、アバタを前に歩かせた場合は操作したキーボードのボタンや移動先の座標などの情報をサーバに送信する。サーバは他のクライアントに対して操作情報を送信する。他の各クライアントは操作情報を受け取り次第、ゲームの状態を更新する。このようにしてゲームの状態は同期される。アバタを操作した後に送信する情報として入力したキーボードのキーやマウス入力を送信する場合や、アバタの移動先の座標を送信する場合などがある。ゲームの状態の同期の方法によってユーザエクスペリエンスやサーバ側の負担、チート耐性が変化する。ここでは、サーバ側の負担が最も大きい場合と最も小さい場合の例を示す。

### 2.2 全てのクライアントの状態を管理する手法

サーバ側の負担が最も大きい場合は図1で示すような全てのクライアントのゲームの状態を管理する手法である。

サーバが全てのゲームの状態を管理し、サーバのみがゲームの状態を直接更新する権限がある。サーバの持つゲームの状態がマスターコピーである。クライアント側はプレイヤーがアバタに対して操作を行ったら、キーボード入力やマウス入力などの情報のみをサーバに送信する。サーバは各クライアントからIO入力情報を受け取り、これに基づいてサーバがゲーム全体の状態を更新する。更新後にサーバの持つゲームの状態を各クライアントに送信する。各クライアントは自身の持つゲームの状態をサーバから送られてきたゲームの状態上で書き直す。サーバの持つゲームの状態

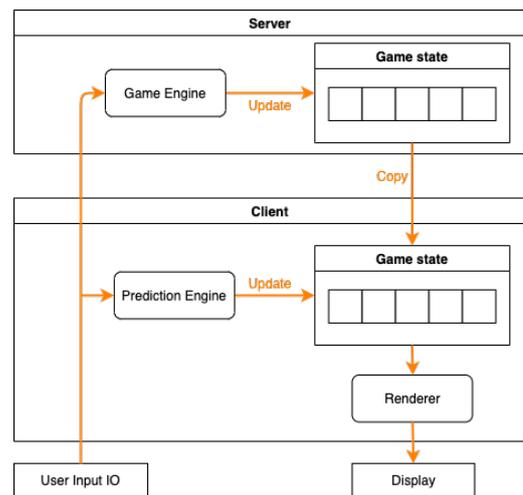


図 1: ゲームの状態を完全に管理するサーバ

がマスターコピーであり、クライアントの持つゲームの状態はレプリカである。このアーキテクチャは SuperTuxKart [5] や Quake [1] というゲームで採用されている。クライアントは単純にサーバ側の状態を受け取るだけである。そのため、サーバ側でゲームの状態の管理が容易にできることから開発が比較的容易になる。さらにクライアントは直接ゲームの状態を変更することはできないため、チートを行うことができない。クライアント側のゲームの状態をしても、サーバや他のクライアントのゲームの状態は変更されないからである。

しかしこのアーキテクチャには2つの問題がある。1つはクライアントの持っているリソースをうまく活用できないことである。最近のクライアントは高機能な GPU や CPU を搭載しているのにもかかわらず、クライアントが行う処理はサーバから受け取ったゲームの状態を画面に表示するだけである。これでは、サーバ側の処理の負荷は高いがクライアント側の処理の負荷はとても低いままである。2つ目はネットワークの遅延による Quality of Experience (QoE) の低下である。インターネットを介してサーバとクライアント間で通信を行うため、自分の行った操作がサーバの持つマスターコピーに反映されるまで時間がかかる。そのため、ファーストパーソンシューティング型やレーシング型ゲームのようなインタラクティブなゲームでは遅延があると QoE を低下させてしまう。例えば、レーシングゲームでは走っている車がカーブに差し掛かっている時に近くを走っている別のプレイヤーの車が曲がるタイミングが遅延によって実際とずれてしまう。相手はすでにハンドルを切っているが自分の画面ではまだ直進しているように見える。相手のハンドル操作のネットワークパケットが自分に届き、自分の画面に反映されると相手の車が瞬間移動したかのように見えてしまう。ファーストパーソンシューティング型ゲームの例ではプレイヤーが敵にライフルで銃弾を当てた時、クライアント側の画面で敵に銃弾が当たった

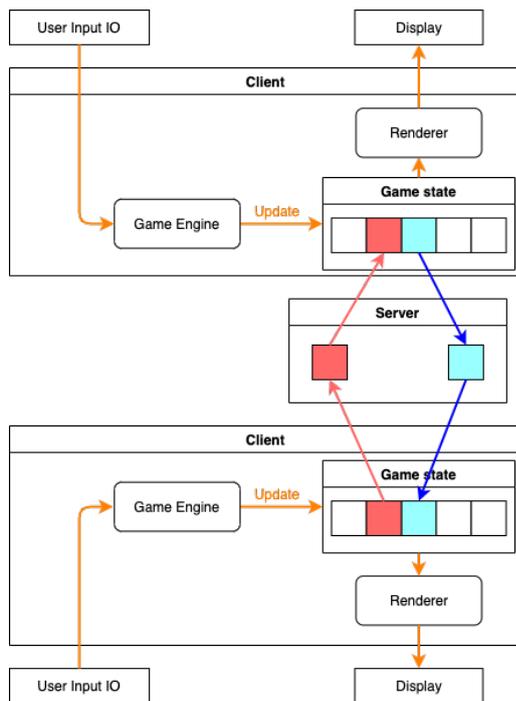


図 2: コア・ロジックをクライアント側にオフロードした場合

ように見えても、サーバ側で処理した結果当たっていないと判断されてしまうことがある。プレイヤーの画面で表示されていることが後から修正されてしまうとゲームの QoE が低下してしまう。近年のオンラインゲームでは遅延があたかも存在しないかのようにプレイヤーに見せるために、クライアント側でサーバが次にどのようなゲームの状態に処理をするのか推測してそれを画面に表示する。けれども、推測の精度は完全ではないため敵が推測の難しい操作を行うとゲームの QoE は低下してしまう。よってこのアーキテクチャではサーバの負担が大きくネットワークの遅延に QoE が大きく左右されてしまう。

### 2.3 ゲームのコア・ロジックをクライアントにオフロードする手法

クライアントのハードウェアリソースの活用とユーザの QoE の改善のためにゲームのコア・ロジックをクライアントにオフロードすることでサーバ側の負担を下げることができる。簡単のためにここでは図 2 に示すようなゲームのコア・ロジックを完全にクライアント側にオフロードする場合について記述する。

プレイヤーがアバタを操作すると、クライアントのマシン上のアバタの状態を表すメモリ上のデータを直接書き換る。書き換えた後のアバタのデータをサーバに送信する。サーバは受け取ったメッセージをそのまま他のクライアントに送信する。他のクライアントはサーバからメッセージを受け取ると、メッセージ内のアバタのデータでゲームの状態を更新する。

これにより、サーバはクライアントからのメッセージを他のクライアントに対してフォワードするだけであるため、負荷を下げるができる。さらに、自分が操作したものが直接反映されるため、後からサーバによって修正されることがなく、QoE が向上する。

しかしこのアーキテクチャはチート耐性が低いという問題がある。これは、他のクライアントから送信されるネットワークパケットに含まれるアバタの状態のデータで上書きすることで、ゲームの状態の更新を行うためである。サーバのマシンのメモリ上のゲームの状態と同期するのではなく、他のクライアントのメモリ上のデータで同期する。そのため、クライアントは自身が送信するネットワークパケットを改竄することでチート行為を行うことができる。また、ネットワークパケットに載っているアバタの状態はメモリ上のアバタの状態であるため、クライアント側のマシンのメモリを改竄することでもチート行為を行うことができる。よって、ネットワークパケットやメモリを改竄することは原理的に可能であるため防ぐことは困難である。

### 2.4 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 [4] 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。

### 2.5 サービスモデル

我々の想定するゲームのサービスモデルはインターネットスケールの複数人対戦型のオンラインゲームである。ゲーム会社がサーバを用意し、マッチメイキングを行う。プレイヤーは自身が所有するマシンをクライアントとしてサーバに接続してゲームを行う。サーバがゲームのマッチメイキングを行い、複数のクライアントが 1 つのゲームマッチに参加する。本論文ではこのようなサービスのゲームのみを対象とする。

## 3. 脅威モデル

本研究で防ぐチートはネットワークパケットの改竄、メモリの改竄によるチート行為である。ゲームアーキテクチャはクライアント側にゲームのコア・ロジックをオフロードしており、クライアントの持つゲームの状態をネッ

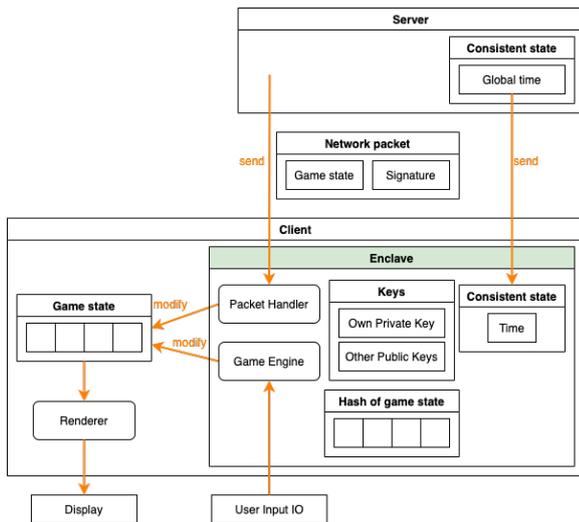


図 3: 「お守り」のアーキテクチャ

ネットワークパケットで送信し、他のクライアントは受け取ったパケットからゲームの状態を更新するという仕組みを採用している。プレイヤー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。OS, VMM, BIOS, メモリバスを不正に操作することにより、ネットワークパケットやクライアント側のマシンのメモリを自由に改竄することができる。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。ゲームのコードにはバグはないとする。

### 3.1 本論文で対象とするチート

オンラインゲームにおけるチートは数多くの種類があるが、本論文で防ぐチートはネットワークパケットやメモリに書き込みアクセスを行なって異常な動作をするようなチートに限定する。メモリやネットワークパケットに読み込みアクセスだけを行い、プレイヤーには本来画面上では知り得ることのないことを知る機密性に関するチート (e.g. Wallhack) は本論文では対象外である。また、グラフィックスやサウンドからの IO を読み取り、画像処理を行い、キーボード入力を不正に自動生成するような (e.g. Aimbot) は対象外である。機密性に関するチート、IO 割り込みによるチートと完全性に関するチートを防ぐ手法は直交である。そのため、機密性に関するチートを防ぐ手法 [11] や IO 割り込みによるチートを防ぐ手法を別に適用することは可能である。

## 4. 「お守り」

我々は「お守り」というオンラインゲームアーキテクチャを提案する。図 3 は「お守り」のアーキテクチャを示す。

「お守り」はクライアント側にゲームのコア・ロジックのほとんどをオフロードしながらゲームの状態の完全性を保証する。クライアント側が送信するパケットやメモリ上

のデータをチータが改竄をしてもチートができない。ネットワークパケットが改竄されないように、TEE 内でパケットにメッセージ認証符号を付けることで、他のプレイヤーからのパケットは Enclave 内で生成されたことを検証できるようにする。メモリ上のゲームの状態のデータの改竄を防ぐためにそのデータをのハッシュ値を TEE 内で管理する。ゲームの状態のデータは TEE 外に配置して良い。これにより、チータがネットワークパケットやメモリを不正に改竄すると TEE が改竄を検知することができる。メッセージ認証コードの生成、検証をするための鍵は TEE 内で管理する。ゲームの状態は TEE の外に配置し、更新を行う関数は TEE の中に配置する。

### 4.1 ネットワークパケットの処理

「お守り」アーキテクチャを適用したゲームでのネットワークパケットの生成はネットワークパケット自体が書き換えられておらず、パケットに含まれるゲームのデータがクライアント側で書き換えられていないということを保証する。「お守り」アーキテクチャを適用したゲームでのネットワークパケットは TEE 内で生成され、自身が操作するアバタやオブジェクトのデータで構成される。ネットワークパケットの生成方法は TEE の外にあるゲームの状態のデータを TEE 内にコピーする。コピーしたデータのハッシュ値を計算し、TEE 内で管理されているハッシュ値と照合する。ハッシュ値が一致したら TEE 外のデータが書き換えられていないということである。ハッシュ値が一致することを確認したらゲームの状態をネットワークパケットにエンコードする。最後に生成したパケットのペイロードに対するメッセージ認証コードを追加する。

### 4.2 ゲーム状態の更新

「お守り」を適用したゲームでは TEE の外にあるゲームのデータが書き換えられていないということを保証する。ゲームの状態の更新をする際は TEE の外にあるデータを TEE 内にコピーする。コピーしたデータのハッシュ値を計算し、TEE 内で管理されているハッシュ値と照合する。ハッシュ値が一致したらゲームの状態を更新するコア・ロジックの関数を実行し、更新する。更新後のデータに対してハッシュ値を計算して、TEE 内で保管する。そして、更新後のデータを TEE の外にあるデータに上書きする。

### 4.3 サーバ側での処理

サーバの役割はマッチメイキングとゲーム中の一貫性の保証を行うことである。マッチメイキングの際にサーバはゲームマッチが始まる前にクライアント側にある TEE のプログラムが正しいものかどうかをアテストーションする。アテストーションを行うことにより、クライアント側の TEE のコードが不正に改竄されていないということ

サーバが確認することができる。ゲーム内のイベントの順番などの一貫性 (consistency) に関わるものはサーバが一括して管理をする。例えば、ゲーム内の時間はサーバが決めた時間を正しいものとするため、サーバが時間の変数のマスターコピーを管理し、そのコピーをクライアントに定期的に送信することで時間を全てのクライアントで同期することができる。

#### 4.4 TEE 外のオブジェクトを TEE 内へコピーする理由

「お守り」ではネットワークパケットを生成する時やゲームの状態を更新する際に必ず TEE 外に配置されているゲームの状態のデータを TEE 内にコピーする。これはハッシュ値のチェックとデータの読み込みアクセスの間の Time of check time of use (TOCTOU) を防ぐためである。TEE 内のコードが TEE 外のメモリにアクセスすることが可能であり、TEE 外のデータを TEE 内にコピーをせずに直接読み込みアクセスをする場合、ハッシュ値のチェックと読み取りアクセスの間の瞬間にメモリを改竄することで不正な改竄をすることができる。そのため、TEE 内のコードが TEE 外のメモリにアクセスすることが可能であったとしても、TEE の外のゲームの状態は必ず TEE 内にコピーしてから使用するべきである。

## 5. 実装

我々はプロトタイプを C++ で開発されているオープンソースの SuperTuxKart というゲームに対して実装する。SuperTuxKart はレーシングゲーム型ゲームである。カートがコースを走り、その順位を競う。それぞれのカートはボウリングボウルを相手に投げたり、バナナを地面に置くなどして他のカートに攻撃することもできる。SuperTuxKart でのゲームの状態は Kart, Track, PowerUp クラスのインスタンスである。各プレイヤ操作するカートは Kart クラスのインスタンスであり、攻撃のアイテムは PowerUp クラスである。競争するレース場は Track クラスで表現されている。

### 5.1 SuperTuxKart のアーキテクチャの変更

オープンソースとして公開されている SuperTuxKart のアーキテクチャはサーバ・クライアント型であり、第 2.2 章で示したアーキテクチャとなっている。そのため、まずは SuperTuxKart のサーバ側で行っていた更新処理をクライアントに移植し、第 2.3 章で示したアーキテクチャにする。今回は Kart と Track 以外のゲームの状態は排除し、Kart と Track の状態のみを更新し続けるゲームとなるように変更する。これにより、Track をカートが走るだけの単純なゲームになる。各クライアントは 1 つの Kart のみを操作する。変更後の SuperTuxKart は以下を無限に繰り返すことで進行する。

- (1) 現在のカートやトラックをレンダリングし、画面に出力する。
- (2) キーボードの入力を元にカートのハンドリングやアクセル、ブレーキ、その他の状態を更新する。
- (3) 次のフレームでのカートのトラック上の位置や速度を物理エンジンを用いて計算し、更新する。
- (4) カートの位置情報の更新の終了後、カートの状態をシリアライズしてバイト配列にしたものをパケットとしてサーバに送信する。

これを繰り返すことでクライアント側のゲームの状態の更新が直接他のクライアントに伝搬するので、サーバの負担を減らしながらゲームを進行することができる。クライアントは別スレッドでサーバからのパケットを待機し、パケットを受け取ったらパケットに含まれるバイト配列から、カートを上書きして更新する。

### 5.2 SuperTuxKart へのお守りの実装方法

以上の変更を加えたプロトタイプに対して、「お守り」を実装した。「お守り」では上記の (2), (3), (4) の処理を Enclave 内で実行する。(2), (3) の処理を実行する手順は以下である。

- (1) Enclave 外にある Kart のインスタンスとキーボード入力情報を Enclave 内にコピーする。
  - (2) Kart のインスタンスのハッシュ値を計算し、Enclave 内に保存されている値と一致するかどうかを検証する。
  - (3) キーボードの入力を元に Kart のハンドリングやアクセル、ブレーキ、その他の状態を更新する。
  - (4) 次のフレームでの Kart の Track 上の位置や速度を物理エンジンを用いて計算し、更新する。
  - (5) Kart のインスタンスのハッシュ値を計算し、Enclave 内に保存する。
  - (6) Kart のインスタンスを Enclave 外にコピーする。
- (4) の処理を実行する手順は以下である。

- (1) Enclave 外にある Kart のインスタンスとキーボード入力情報を Enclave 内にコピーする。
- (2) Kart のインスタンスのハッシュ値を計算し、Enclave 内に保存されている値と一致するかどうかを検証する。
- (3) Kart のインスタンスをシリアライズしてバイト配列にする。
- (4) バイト配列に CMAC で認証コードを計算する。
- (5) バイト配列と認証コードをパケットとしてサーバに送信する。

ネットワークパケットを待機している別のスレッドでは以下を行う。

- (1) Enclave 外にある Kart のインスタンスとキーボード入力情報を Enclave 内にコピーする。
- (2) Kart のインスタンスのハッシュ値を計算し、Enclave 内に保存されている値と一致するかどうかを検証する。

(3) パケットから Kart のバイト配列と認証コードを取り出し, CMAC の共通鍵で認証コードを検証する.

(4) パケットから kart のバイト配列か Kart のインスタンスを再構築し, Enclave 外の Kart のインスタンスに上書きする.

このような方法で, Enclave 外の Kart の状態の完全性を保護している. Track のインスタンスのサイズは非常に大きく, ゲームの更新時に状態が変更されることが少ないため, Enclave の中にインスタンスを保管しているが, Enclave の外にコピーすることはしていない.

## 6. 評価

**実験環境.** 我々は SuperTuxKart 上に実装した「お守り」をラップトップマシンに実装する. マシンの CPU は Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz で SGX Enclave のサイズは 256MB, OS は Ubuntu18.04, RAM は 16GB である.

我々は SGX を使用していない SuperTuxKart と使用していないものの性能を比較する. 本当の SuperTuxKart には走行中のカートがアイテムを投げたり, 地面に落とすなどの要素や,トラック上の特定の地面を踏むとカートが加速するなどの様々な仕掛けがある. けれども, 「お守り」ではこのような複雑な要素を実装していない. そのため, なるべく公平な比較をするために, SuperTuxKart のカートがトラックを走る以外に複雑な機能は全て除く. これにより, 性能の違いは「お守り」の影響のみになるように設定する.

### 6.1 1 フレームの処理時間

クライアントが1つの場合でトラックを5周走ったときの1フレームの処理時間を計測する. 1フレームの処理時間の中にはゲームの状態の更新, ネットワークパケットの生成, 送信, 画面の描画処理が含まれる. プロトタイプではこれらを1スレッドで順番に繰り返し実行している. 画面の描画処理の速度はトラック上のカートの位置によって大きく変化する. 周りに木や建物などのオブジェクトがたくさんある場所では画面のレンダリング処理の時間が大きくなる. そのため, 我々はスタート地点からの距離と1フレームの処理時間を記録する. 実験のカートの操作は手動で行い, SGX を使った場合と使っていない場合でなるべく同じ軌跡を辿るように走ることによってカートの近くにあるトラックのオブジェクトの数の影響に差が出ないようにする.

図4に計測結果を示す. 1フレームあたりの処理時間はお守りではわずかに「お守り」の方が高いが, どちらもほぼ同じ処理速度と言える. 処理時間の平均は Baseline では 23.3 ms, 「お守り」では 23.9 ms であり, 「お守り」を使用すると 0.6 ms 処理時間が増える. 平均処理時間を FPS に換算すると, Baseline では 44.6, 「お守り」では 43.5 で

表 1: ゲームの状態の更新の処理時間

	MAX	MIN	AVG
Baseline	0.204	0.0066	0.0148
Omamori	0.631	0.090	0.155

表 2: ネットワークパケットの生成の処理時間

	MAX	MIN	AVG
Baseline	0.0665	0.00718	0.0147
Omamori	0.201	0.0672	0.0976

あり, 人の体感では違いがわからない程度にパフォーマンスが低下した.

### 6.2 ゲームの状態の更新の処理時間

ゲームの状態の更新のみの処理時間を計測する. ゲームの状態の処理にはユーザの入力をカートの速度に反映させる処理や, Track と Kart の物理エンジンで処理して次の位置の計算が含まれる. 「お守り」を使用することによるオーバーヘッドは Untrusted 側にある Kart のインスタンスを Enclave 内外にコピーする処理と, Hash 値の照合と更新, Enclave 内で演算をすることによる計算速度の低下がある. 表1に結果を示す. 「お守り」を使用することによってゲームの状態の更新の処理時間は平均で 10.5 倍に増加した. しかし, 増加幅は 0.14 ms 程度であるため, ゲームの実行には大きな影響はない.

### 6.3 ネットワークパケットの生成の処理時間

ネットワークパケットの生成の処理時間を計測する. パケットの生成の処理はカートの状態をバイトの配列にシリアライズする処理である. 「お守り」を使用することによるオーバーヘッドは Untrusted 側にある Kart のインスタンスを Enclave 内にコピーする処理, Hash 値の照合, CMAC での認証コードの生成, Enclave 内で計算をすることによる計算速度の低下がある. 表2に結果を示す. 「お守り」を使用することによってパケットの生成の処理時間は平均で 6.6 倍に増加した. しかし, 増加幅は 0.083 ms であるため, ゲームの進行には影響はない.

## 7. 関連研究

BlackMirror[11] はクライアントサーバ型のオンラインゲームアーキテクチャの Wallhack というゲームの機密性に関するチートを防ぐ. Wallhack とはプレイヤーからは本来見ることのできない画面外にいる敵を見るというチートである. このチートはメモリに載っている本来は見えないデータを不正に覗き見ることで行われる. BlackMirror は敵に関するデータ全体を Enclave 内で保管し, 画面に映る敵のみを Enclave 外にコピーすることで Wallhack を防ぐ. BlackMirror は機密性に関するチートを防ぐが, 完

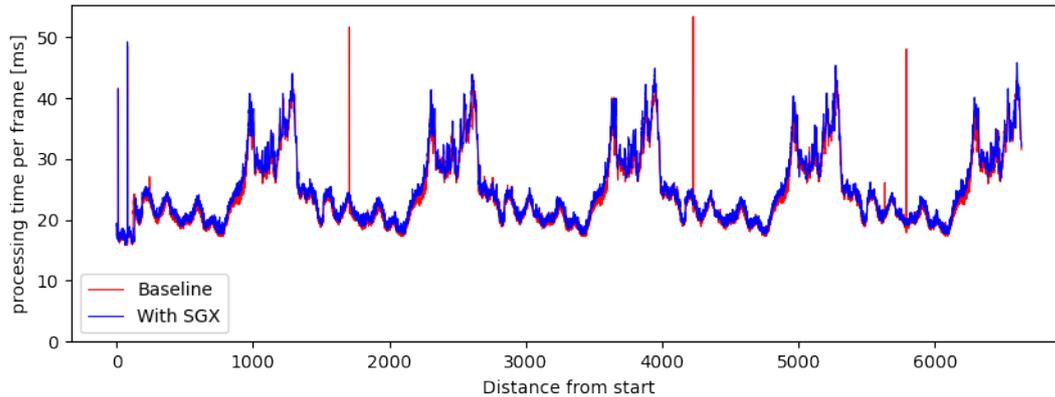


図 4: クライアントが 1 人の場合での走行距離と 1 フレームあたりの処理時間の関係

全性に関するチートを防ぐものではない。BlackMirror と「お守り」は直交する。

LambdaContainers[10] はモバイルゲームでの完全性に関するチートを防ぐ。ユーザごとに対応するコンテナをサーバ側で作成し、ユーザの環境をエミュレートし、ユーザのアプリの不正な挙動を検知する。しかし、サーバ側の負荷が高くなってしまふ。

Bauman[6] は SGX でゲームを保護する方法の 1 つを紹介している。しかし、Bauman は Enclave にゲーム全体を入れることでゲームの DRM のみを保護している。具体的なチートの保護については今後の課題としている。

## 8. 結論

オンラインゲームではスケーラビリティのためにクライアント側にコア・ロジックをオフロードしている。けれども、クライアント側のコードやデータを不正に改竄することでチート行為を行うことができる。そのため、オンラインゲーム産業ではチート行為が問題となっている。我々はクライアントにオフロードされたコードやデータの完全性を TEE を使って保護することを提案する。実際に「お守り」を SuperTuxKart というゲームに Intel SGX を使って実装し、性能を示した。

**謝辞** 本研究は JSPS 若手研究 JP21K17726 の助成を受けたものです。

## 参考文献

- [1] : IOQUAKE3, <https://ioquake3.org>.
- [2] : A Letter from the Anti-Cheat Team, PUBG Corporation (online), available from <https://www.pubg.com/2019/02/26/a-letter-from-the-anti-cheat-team/> (accessed 2021-04-20).
- [3] : POBG The Original Battlegrounds, <https://www.pubg.com>.
- [4] : Steam Hardware & Software Survey: December 2019, <https://store.steampowered.com/hwsurvey/processormfg/>.

- [5] : SuperTuxKart, [https://supertuxkart.net/Main\\_Page](https://supertuxkart.net/Main_Page).
- [6] Bauman, E. and Lin, Z.: A Case for Protecting Computer Games With SGX, *Proceedings of the 1st Workshop on System Software for Trusted Execution, Sys-TEX '16*, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/3007788.3007792 (2016).
- [7] Delgado, G.: Unlock Your Full Potential - How Higher Frame Rates Can Give You An Edge In Battle Royale Games, Nvidia Corporation (online), available from <https://www.nvidia.com/en-us/geforce/news/geforce-gives-you-the-edge-in-battle-royale/> (accessed 2021-04-20).
- [8] Intel: Software Guard Extensions Programming Reference, Ref. 329298-002US, <https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf> (2014).
- [9] Jang, T.: Dev Letter: Anti-Cheat & Performance Plans for 2020, PUBG Corporation (online), available from <https://www.pubg.com/en-us/2020/04/21/dev-letter-anti-cheat-performance-plans-for-2020/> (accessed 2021-04-20).
- [10] 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).
- [11] Park, S., Ahmad, A. and Lee, B.: BlackMirror: Preventing Wallhacks in 3D Online FPS Games, *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS '20*, New York, NY, USA, Association for Computing Machinery, p. 987–1000 (online), DOI: 10.1145/3372297.3417890 (2020).