BlackMirror: Preventing Wallhacks in 3D Online FPS Games

Seonghyun Park
Seoul National University
shpark95@snu.ac.kr

Adil Ahmad
Purdue University
ahmad37@purdue.edu

Byoungyoung Lee*
Seoul National University
byoungyoung@snu.ac.kr

ABSTRACT

Online gaming, with a reported 152 billion US dollar market, is immensely popular today. One of the critical issues in multiplayer online games is cheating, in which a player uses an illegal methodology to create an advantage beyond honest game play. For example, wallhacks, the main focus of this work, animate enemy objects on a cheating player's screen, despite being actually hidden behind walls (or other occluding objects). Since such cheats discourage honest players and cause game companies to lose revenue, gaming companies deploy mitigation solutions alongside game applications on the player's machine. However, their solutions are fundamentally flawed since they are deployed on a machine where the attacker has absolute control.

This paper presents BlackMirror, a new game design with a trusted execution environment, Intel SGX. Leveraging strong data isolation guarantees provided by SGX, BlackMirror addresses the root cause of wallhacks by storing sensitive game data within an SGX-protected region. BlackMirror overcomes various challenges in achieving its goal including partitioning game client to avoid SGX's memory limitations, as well as cross-checking inputs provided by untrusted keyboard and mouse. Furthermore, BlackMirror supports GPU-based 3D rendering by performing highly-accurate visibility testing and disclosing sensitive data only when it is required in a given game scene. We protect Quake II using BlackMirror, and our evaluation results demonstrate that BlackMirror-protected Quake II is fully functional and secure. More specifically, BlackMirror incurs 0.57 ms per-frame delays on average, which meets modern game's performance requirements. On the other hand, the secure baseline design using software-only rendering incurs 25 ms per-frame delays on average, signifying the efficient yet secure design of BlackMirror.

CCS CONCEPTS

• **Security and privacy** → **Domain-specific security and privacy architectures**; **Trusted computing**.

KEYWORDS

multi-player games; wallhacks; Intel SGX

*Corresponding author

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417890

ACM Reference Format:
Seonghyun Park, Adil Ahmad, and Byoungyoung Lee. 2020. BlackMirror: Preventing Wallhacks in 3D Online FPS Games. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3372297.3417890

1 INTRODUCTION

Online gaming is one of the most popular entertainment platforms in the world. With the soaring popularity of electronic sports, competing in video games has also become a mainstream profession for many people—it is reported that online game markets generated 152 billion US dollars [1]. Among online games, first-person shooter (FPS) games are an incredibly popular genre. In the FPS game, a player is centered with his weapon, shown a first-person perspective onto a virtual game scene. The common goal of the FPS game is to find and eliminate the other players (or teams). Common FPS games include Doom, Quake, Overwatch, Fortnite, and PUBG.

However, due to the competitive nature of online games, many players are intrigued to use illegal methods (or cheats) to create an advantage beyond honest players. Such cheating behaviors are not inconsequential in today's online games—according to a Forbes report [2], a total of 37% gamers have cheated before. With such behaviors, game play is not fair, severely ruining fun for participating players and integrity of competitive games. The same report mentions that cheating severely impacts the gaming experience of players—88% of gamers stated that they experienced an unfair game due to the cheating.

Cheating in online games is very serious for game companies whose revenue stream heavily depends on player satisfaction. Therefore, many gaming companies are deploying games with anti-cheat solutions [3–5]. These solutions, operating either at the user-level or privileged levels, are designed to detect cheating attempts. However, these solutions are ad-hoc (i.e., result in arms races between cheaters and game developers), and/or require installing proprietary (closed-source) kernel modules or drivers on the player's machine. Importantly, these solutions do not solve the fundamental issue surrounding cheating—the attacker has complete control on their machine and can compromise game clients, extract sensitive information, and bypass anti-cheat solutions.

Particularly focusing on FPS games, wallhack [6] are one of the most commonly-used cheats. In particular, the common exploit pattern of wallhacks is to animate enemy players hiding behind walls (or other occluded entities) in front of the player. Therefore, the dishonest player can easily spot an unsuspecting enemy player without being visible on the other player's screen. The root cause behind wallhacks is that the dishonest player has access to states belonging to sensitive entities (e.g., opponent's position). In particular, the dishonest player locate relevant states in game clients [7], eavesdrop on or tamper with these states as they are transmitted

through the CPU-GPU communication channel [8], or tamper with the GPU computation itself [9].

In this paper, we present BlackMirror, a system that prevents wallhacks in multiplayer online FPS games using Trusted Execution Environments (TEEs). In particular, BlackMirror protects security-sensitive data (e.g., opponent entities) and only permits access if such data is required to be rendered in the game scene. In this regard, the key idea behind BlackMirror is to strictly safeguard such sensitive data with a TEE, particularly Intel SGX. Intel SGX is a good fit since it provides hardware-assisted isolation with confidentiality and integrity guarantees, as well as remote attestation capabilities to attest the correctness on untrusted player machines. Furthermore, SGX is already deployed on popular gaming processors, i.e., all Intel desktop processors.

However, realizing BlackMirror is not straightforward. First, it is unclear what sensitive data should be stored within the SGX-protected region, an enclave. From a security stand-point, it is best to store everything inside the enclave, but due to memory limitations of SGX enclaves §5, this will incur prohibitive overhead. Second, BlackMirror should maintain compatibility with existing game client functionality while ensuring correctness and security. In particular, existing game clients update internal gaming contexts (e.g., movement of and events from all players in the game) using server messages and keyboard/mouse inputs. Third, a game client should leverage GPU to accelerate 3D rendering performance, but the protection realm of SGX does not include the GPU. While software-only rendering is possible, it is remarkably slow (as we demonstrated in §9.2).

To solve the aforementioned challenges, BlackMirror features the following design characteristics. First, BlackMirror carefully partitions the game client into trusted (i.e., enclave) and untrusted components, ensuring security while reducing enclave memory consumption. In particular, BlackMirror maintains a thin enclave-layer that stores only data relevant to sensitive game entities (i.e., entities directly involved in determining the view of the player). In particular, such data includes: a game state related to enemy players, occluding entities such as walls and other supporting data such as geometry information related to these sensitive entities.

Second, BlackMirror provides in-enclave functionality for secure updates to the protected sensitive entities. In particular, these updates are received from the untrusted world through (a) server's update messages received through attacker-controlled network interfaces, and (b) potentially malicious inputs received from untrusted I/O devices such as keyboard and mouse. To solve these issues, BlackMirror creates a secure channel with the game server terminating within the enclave, and synchronizes all provided updates with the information sent by the attacker to the game server. As a result, BlackMirror is able to detect discrepancy in provided inputs or forces the attacker to reveal itself to other players if they attempt a wallhack.

Lastly, to support vital hardware-acceleration using GPUs without compromising on security, BlackMirror implements a trusted visibility test within the enclave. For each frame, BlackMirror performs visibility-testing to determine which of the requested entities by the untrusted game client should be visible on the player's screen. Using trusted information stored regarding sensitive entities, BlackMirror is able to determine with high accuracy whether this entity should be visible in this frame. Furthermore, thanks to recent advances in floating-point computation on CPUs (e.g., AVX and SIMD instructions), BlackMirror can perform this rapidly and in a scalable manner.

We implemented BlackMirror-protected Quake II, an open source 3D FPS game. According to our evaluation, BlackMirror-protected Quake II demonstrated that it is not only fully functional but also secure. To be more specific, BlackMirror incurs 0.57 ms per-frame delays on average, which meets modern game's performance requirements (i.e., considering 60 fps game, the time gap between frames is 16 ms). On the other hand, the secure baseline design using software-only rendering incurs 25 ms per-frame delays on average, signifying the efficient yet secure design of BlackMirror. From the security perspective of BlackMirror, the accuracy of visibility testing (i.e., declassification accuracy) is at least 97%, suggesting that most of sensitive entities are accordingly secured against cheaters.

To summarize, this paper makes the following contributions:

- **Design**. Understanding the root cause of wallhacks, we designed BlackMirror, an SGX-based game client to fundamentally prevent wallhacks. In order to leverage GPU-based rendering while preserving SGX's security assurance, BlackMirror designs an in-enclave visibility testing and declassify safety-confirmed data to GPU.
- **Implementation**. We implemented BlackMirror-protected Quake II, an open source 3D FPS game. Our evaluation confirms that BlackMirror-protected Quake II is fully functional while meeting gaming user's experiences.
- **Security Analysis**. We thoroughly analyze the security aspect of BlackMirror. We exhaustively consider all possible attacks that can be launched by a wall-hack motivated attackers, and analyze why BlackMirror is secure against those attacks.

2 BACKGROUND

This section provides background information relevant to the design of our system, BlackMirror. We first describe online game architecture (§2.1). Next we explain how the rendering pipelines in the game operate (§2.2), and then describe Intel Software Guard eXtensions (SGX) (§2.3).

2.1 Multiplayer FPS Game Architecture

Modern multiplayer first-person shooter (FPS) games [10] operate on the client-server architecture (shown in Figure 1). In particular, multiple clients connect to a dedicated server, which is usually remote and operated by the game operating companies.

The game state is a set of entity states, where an entity refer to each object in the game. Such an entity can be player's characters, supplies and environmental objects. Each entity has state such as the 3D origin (i.e., its coordinates in 3D space), angles (e.g., the direction that an entity is facing), etc. Figure 2 shows an example entity state from Quake 2 [11].

The server and clients maintain their own copy of a game state, which we refer to a global game state in the server and a local game state in clients, and each has a different mechanism in updating the game state.

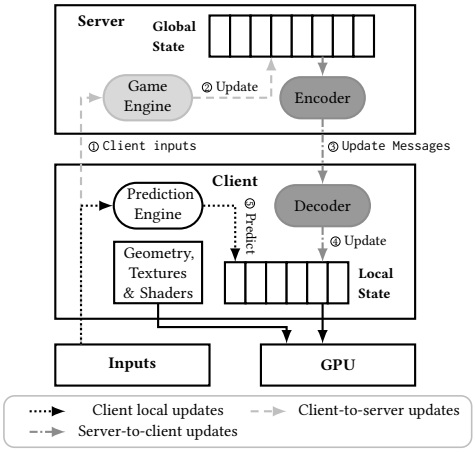


Figure 1: Client-server architecture of 3D online games

A Global Game State in Server. The server plays the role of a central authority that stores the *global game state*. In particular, the server is acting like an event-driven system, updating its state (②) in response to client inputs (e.g., mouse clicks and keyboard inputs as shown in ①) that are captured within a certain interval, called a *frame*. For example, if the client moves the mouse, the command is sent to the server, which updates the view angle of this player's character in its corresponding entity.

A Local Game State in Client. A game client holds a local game states to render the game scenes. In modern games, there are two major sources of local state changes: 1) Server update message and 2) Local state prediction.

The server periodically sends update messages to all clients (③), which indicates how local states should be updated in order to synchronize with the global state. Then the client updates its local state with the messages (④). Leveraging this synchronization method alone has an issue to support both smooth gameplay experience and low network bandwidth. To support smooth gameplay experience, the update message should be sent very frequently because the scene rendering should be performed based on a very up-to-date game state. However, such frequent updates impose severe network bandwidth. In order to address this issue, modern games employ the local prediction method that we describe next.

The local state prediction is to predict the local game state based on the player's input and elapsed time from the last frame. (❺). For instance, the client records the fraction time, when the key was down, multiply a speed value, and add it to the previous position of the player. Predicting is crucial for providing smoother scene update, since the update messages from the server arrives less frequently than the frame refresh rate (e.g., 60 fps) [10]. Therefore, by combining client-side prediction with periodic state updates, modern games are able to provide smoother gameplay experiences, while keeping network bandwidth low.

2.2 Game Rendering Pipeline

In this subsection, we describe how a client render the game on there machine so as to display the game scene to the display.

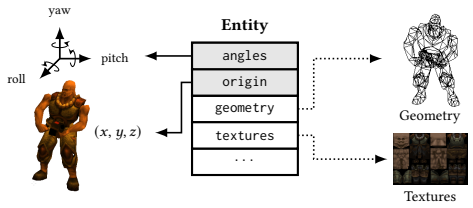


Figure 2: An example entity in Quake II [11]

Coordinate Systems in 3D Rendering Pipeline. In general, rendering is the process of synthesizing multiple 3D models, thereby generating a 2D scene shown to the user's display. During the rendering process, a sequence of space transformation are performed, where each transformation transforms from one space to the next: 1) Local space; 2) World space; 3) Clip space; and 4) Screen space. Local space is the 3D space that each entity is represented with its own local coordinates (i.e., each entity has an independent 3D coordinate system). Then the local space is transformed into the world space. The world space is the 3D space that all the entities are projected into the game world space (i.e., the game map). So in this world space, all the entities are positioned in a single shared 3D coordinate system, which represents the 3D game world. Next, the world space is transformed into the clip space, which is the 3D space that is captured by the camera (i.e., the self-player). Finally, the clip space is transformed into the screen space, which corresponds to the final 2D scene that will be shown to the user's display. In the followings, we describe the rendering process, especially focusing on the different computational roles of CPU and GPU.

Stage 1: Rendering Preparation on CPU. A general idea of graphics rendering pipelines is that CPU performs light-weight tasks and GPU performs the rest heavy-weight computation. In particular, the game client, on CPU, prepares the transformation matrix (e.g., `worldToClip` and `localToWorld`) from the entity states. Then such matrices and auxiliary data (e.g., model and shaders) are sent to the GPU, which performs expensive computations including matrix computation and rasterization.

In the rendering preparation step, the client prepares i) two transformation matrix, `worldToClip` and `localToWorld`; and ii) auxiliary rendering data (e.g., models and shaders). The `worldToClip` is a matrix that transforms an entity in the world space into an entity in the clip space. This matrix is derived using the origin and angles of the self-player entity, and the field-of-view, which is an angle that represents the visible range. (❶) The `localToWorld` is a matrix transforming an entity in the local space into an entity in the world space. This matrix is calculated using the origin and angles of each entity that are being transformed. (❷)

The client also prepares the auxiliary rendering data (such as models and shaders) associated with each entity. In particular, the geometry stores the the shape of an entity in local space, and the texture represents the color of the surface (see Figure 2). Additionally, the shaders, namely the vertex shader and the fragment shader, defines the computation that is run on the GPU.

Finally, two transformation matrices and the auxiliary rendering data is sent to the GPU (❸), which starts the next rendering pipeline stage, geometry pipeline.

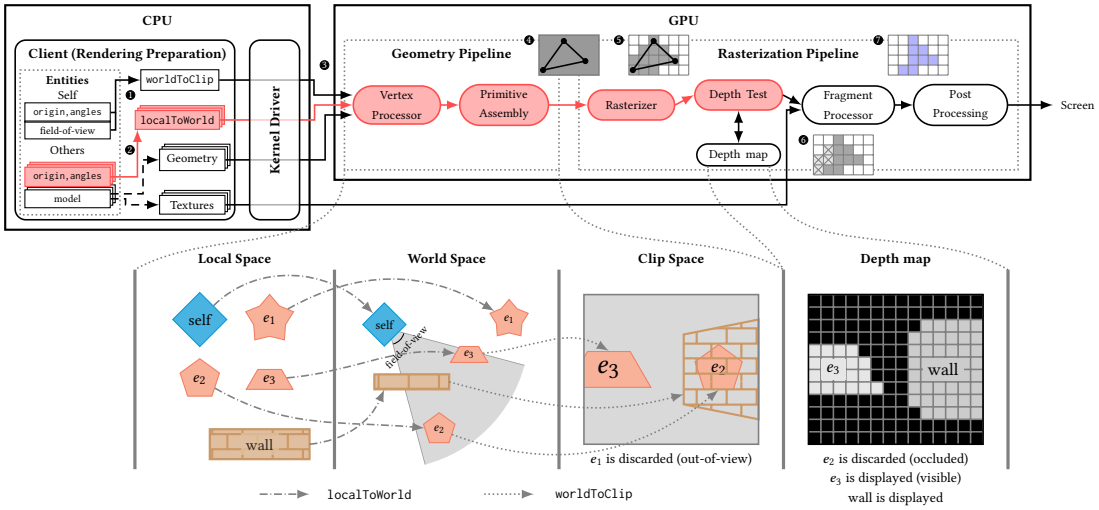


Figure 3: Overview of rendering pipeline (top) and corresponding transformations in coordinate systems (bottom)

Stage 2: Geometry Pipeline on GPU. Upon receiving the request from CPU, GPU starts the geometry pipeline stage. The goal of this stage is to apply a series of coordinate transform to entities, therefore transforming them from the local space to the clip space.

The most crucial units in this stage is the vertex processor, which applies the transformations defined in the vertex shader to input geometry. A typcical vertex shader define the following transformation, therefore transforming from the local space into clip space: `v' = worldToClip * localToWorld * v`.

As a result, entities in the local space, e_1, e_2, e_3 and the wall are transformed into the clip space as shown in Figure 3. Notably, the z value in the clip space represents the distance between an entity and the camera, which we refer to depth value. Next, the primitive assembler groups vertices into geometric primitives such as triangles.

After completing the geometry pipeline, vertices in clip space are passed to the next stage, the rasterization pipeline, with additional information about their grouping.

Stage 3: Rasterization Pipeline on GPU. This stage takes an entity in the clip space and determines i) which portion of an entity is shown to the 2D screen, and ii) its final color.

First of all, the rasterizer transforms input clip space into a screen space, where (x, y) coordinates correspond to the index of pixel in the final 2D scene (❹ → ❺). Then, the GPU also computes missing attribute values of each pixel (e.g., the depth value) that overlaps with an entity. Recall that previously the GPU only knows the attributes of points in the geometry, not the surfaces.

Next, the depth test stage takes an entity in the screen space and determines visible pixels. The primary goal of depth testing is to keep the correctness of the rendered scene, e.g, entities behind the wall should not be drawn. To this end, the GPU keeps an internal data structure, depth map, which keeps the closest depth values that have been rendered. For example, imagine the wall in Figure 3 is rendered in advance to e_2, and the GPU is now about to render it. At this point, the depth values of the wall is already stored in the depth map, and these are closer than the depth values of e_2. Consequently, the GPU discards the entity e_2, thus the following stages are not performed for e_2. (❻)

Lastly, the fragment processor runs the fragment shader code for all visible visible entities to compute their color. In particular, it performs texture mapping and lighting to compute the color for each pixel (❼), which comprises expensive operations.

2.3 Intel Software Guard eXtensions (SGX)

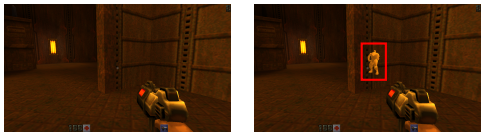
Intel SGX [12, 13] allows a user-level process to create its own isolated execution environment, called enclave, which is protected against all privileged softwares including OS and hypervisors. Enclaves reside in a pre-determined (at boot-time) location of physical memory called the Enclave Page Cache (or EPC). However, Intel SGX has various well-known limitations.

In particular, SGX does not provide trusted I/O paths off-the-shelf. Enclaves can seal [13] persistent data before storing it on the disk and encrypt network communication, yet there are scenarios where sealing is not helpful. For example, GPU communication is in plaintext, due to lack of encrypted communication capabilities within commodity GPUs. On the other hand, data from input devices such as a keyboard and a mouse are not protected, either. Although existing research has considered this problem, they require complicated changes to the software or hardware ecosystem of SGX, such as using a trusted hypervisor [14], extra hardware [15, 16] or hardware changes [17, 18].

3 THREAT MODEL AND ATTACK SUMMARY

3.1 Threat Model

We assume the attackers control their machine including the peripheral devices, and privileged software including OS and hypervisors. In particular, an attacker i) is capable of monitoring and modifying the client memory, ii) eavesdropping on and tampering with the communication between the CPU and the GPU and/or iii) read or write kernel memory.



(a) Benign view (b) Wallhack view

Figure 4: An example of wallhacks. The opponent entity behind the wall is rendered to the scene, which is highlighted with the red box in Figure 4b.

We assume that game server is correctly implemented, run by a trusted party (e.g., a game company), and is not colluding with a player. This is also a fair assumption since the game companies have an incentive to maintain their reputation amongst players.

On the other hands, we leave the following as out-of-scope: i) side-channels and micro-architectural attacks and ii) software vulnerability in enclaves. SGX has been a famous target for various side-channel attackers and micro-architectural attacks [19–25]. In this work, we are not dealing with these attacks. Furthermore, enclave code possibly contains software vulnerabilities, e.g. buffer overflow [26, 27], and existing solutions [28, 29] should be used to prevent them.

3.2 Wallhack Cheats

The attacker wants to perform better than other players in competitive online games. To achieve this goal, the attacker uses the infamous wallhack cheats [30], which result in hidden entities (e.g., rival players behind a wall or out-of-view) appearing on the attacker's screen. For example, consider entities in the world space of Figure 3, where only e_3 is visible, and entity e_2 and e_3 are hidden at this point (e_2 is blocked by the wall and e_1 is out-of-view).

Root Cause of Wallhacks. The root cause of wallhacks originates from an inherent feature of current online multiplayer games, i.e., the client application holds more states than required to render a scene. To elaborate, the server sends information of non-visible opponent entities to the client to improve gameplay (§2.1). As far as honest players are concerned, these states are inconsequential since they will not be rendered in the current scene. However, the attackers can exploit these states to perform a wallhack.

Unfortunately, filtering unauthorized information at the server has various issues. In particular, the server has to perform additional computation for each player, especially if many clients are connected to the server at the same time. Additionally, filtering information at the server can result in unpredictable gameplay. For example, the user may observe visual glitches (e.g., opponents appearing out of thin air), if the server-side filtering is too strict. Furthermore, the server lags behind the client due to network delays, and therefore server-side filtering cannot be as precise as the client-side solutions. As a result, server-side filtering ends up being too conservative.

Figure 3 describes how unauthorized states propagate within the rendering pipeline, until they are discarded. In particular, the client application blindly runs rendering pipeline without performing visibility-testing. These states propagate in the memory and the GPU, until depth-testing is performed and they are discarded.

Attack Surfaces. In the following, we summarize possible attack surfaces of wallhacks, based on investigation of publicly available wallhacks [8, 9, 31–33]. Recall that the client is provided with states of non-visible entities and it blindly issues rendering calls for them, therefore sensitive information propagates from the client memory to the GPU.

Game Client. The attacker can read sensitive information from client memory, or modify the client to tamper with the its execution. Typically attackers attempt to locate the opponents' entities in the memory [7, 34], read their positions and overlaying the scene with these information [35] Alternatively, the attackers may exploit existing code to render extra information on the scene. [33]

Communication between CPU and GPU. An attacker can exploit the kernel subsystem or communication between the CPU and the GPU to leak secrets and manipulate rendering result. The rendering requests issued by the client are mediated by the kernel subsystem or the driver before transmission to the GPU. For example, an attacker tampers with the rendering process by hooking the rendering functions [8, 9].

GPU Computation. Lastly, an attacker can undermine the integrity of computations performed on the GPU. For example, an attacker-controlled driver may modify the shader, or reject certain rendering requests (e.g., skip rendering the walls). Consequently, if the GPU issues a rendering requests for an opponent entity behind the walls, the opponent entity will appear on the screen. On the other hand, an attacker may fool the GPU to disable depth testing of certain entities [8], therefore drawing the entities over the wall.

4 LIMITATIONS OF CURRENT ANTI-CHEAT

In the following, we describe the shortcomings of modern anti-cheat software, which we try to overcome with our proposed design.

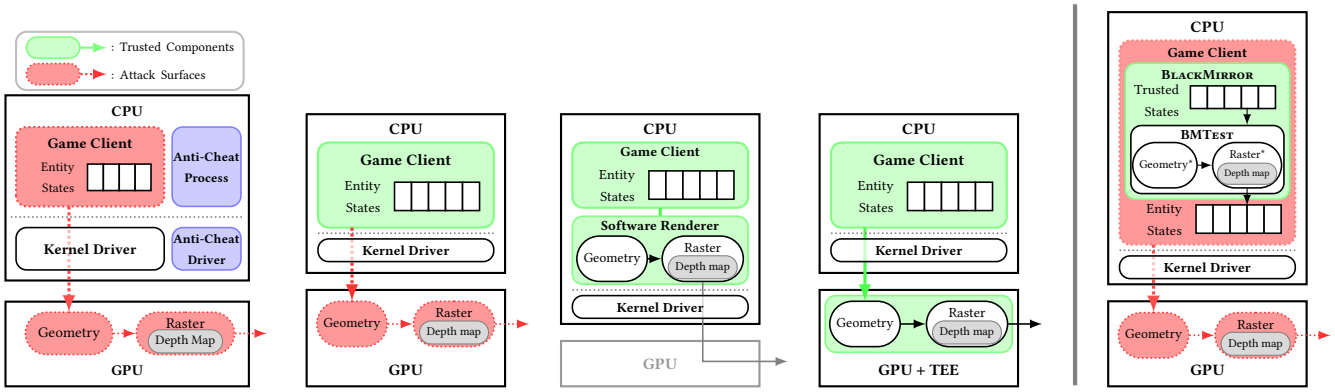
Commercial Anti-Cheat Software. Many commercial games are deployed with third-party anti-software [3–5, 38] or their own anti-cheat solutions [39]. However, we observe that these approaches fail to provide sufficient security guarantees, resulting in an arms race between attackers and anti-cheat developers.

As shown in Figure 5a, anti-cheat solutions are commonly deployed as an external process, and an anti-cheat kernel driver [3, 6]. In particular, anti-cheat software continuously challenges the player's game client to prove there is no violations. For example, it looks for malicious contents (i.e., known cheats) loaded on the client memory, detects hypervisors and manipulated control flow [40].

Despite their efforts, tech-savvy cheaters keep bypassing such defenses and the anti-cheat developers keep patching their mitigation solutions in response. Even worse, such an arms race has moved to the kernel space, since modern cheats operates in kernel or hypervisor to avoid anti-cheat and the game companies deploy specific anti-cheat kernel modules as countermeasures. In consequence, the users are enforced to run kernel-level anti-cheat modules to play the game, and unfortunately this opens up new attack vectors [6, 30].

5 WALLHACK PREVENTION USING TEEs

In this section, we explore potential designs to protect games against wallhacks by leveraging the confidentiality and integrity protections provided by TEEs.



(a) Anti-Cheat Software [3, 4] (b) Client in Enclave [36] (c) Software Renderer [37] (d) Trusted GPU [17, 18] (e) Our Approach

Figure 5: Comparisons between Our Approach, Anti-Cheat Software and Alternative Approaches Using Trusted Execution

D1. Enclosing Game Client within Enclave. The most straightforward design is to enclose the game client within an enclave (Figure 5b). In fact, this design is identical to typical use-cases of TEEs to augment application security—running entire applications within enclaves (e.g., using Library OS [36, 41]). As far as security is concerned, this design prevents access to sensitive entities (i.e., violating confidentiality), and modifying the original client code (i.e., violating integrity).

However, by nature of existing TEEs, this design has a critical limitation, i.e., it only protects the game client within the enclave, and interfaces between the enclave and rest of the system are left unprotected. In particular, the CPU-GPU channel and GPU computation (refer Figure 5b) are still possible wallhack attack surfaces. Therefore, this approach by itself is insufficient.

D2. Software-only Rendering inside Enclave. Improving on D1, this design implements the entire rendering pipeline within an enclave. Therefore, this design is more secure as compared to D1, because the rendering pipelines are now placed within the protection boundary of the TEE. However, this design has to perform the software-only rendering, so it has to abandon crucial GPU-acceleration for rendering computation. Therefore, the rendering performance of such a solution is unimaginably slow (§9.2), and incapable of meeting current gameplay requirements.

D3. Rendering with Trusted GPU. Previous works on trusted hardware [15–18] have introduced promising ways of extending trusted execution to external accelerators or augmenting SGX using I/O protection. Unfortunately, these approaches require hardware changes to existing architectures/devices, which would be challenging to achieve given the diversity of trusted execution environments and devices commonly-used.

6 DESIGN OF BLACKMIRROR

In this paper, we present an alternative approach to defeating wallhacks using trusted execution. In particular, we present an anti-cheat system, where players do not have to install untrusted proprietary kernel modules on their machines and neither does the game company have to deploy insecure obfuscation techniques to hide sensitive information.

The key idea behind BlackMirror is to provision the game application with an SGX enclave and store all sensitive game entities within the enclave, and therefore, inaccessible to the potentially dishonest player. More specifically, sensitive game entities (e.g., enemy characters) reside in the enclave even when they should not be visible to the player. To maintain compatibility with legacy game engines, BlackMirror performs in-enclave updates to these states based on secure messages from the server and player inputs.

Then, BlackMirror performs in-enclave visibility testing on each frame to determine the visibility of entities requested by the untrusted game client, in the current frame. If an entity is found to be visible, BlackMirror declassifies such an entity—i.e., the current state of the entity is provided to the untrusted client, so that it can be processed by GPU. Otherwise, BlackMirror keeps the states secure in the enclave. Note that this does not harm the original functionality of our rendering pipeline, because non-visible entities will not be processed anyways.

In the remaining, we first describe how BlackMirror bootstraps on the machine and creates a secure connection with the server (§6.1). Then we describe which entities are sensitive and thus stored within an enclave (§6.2). Next, we explain how BlackMirror updates the entity states within an enclave to maintain the game functionality (§6.3). Lastly, we describe how BlackMirror performs in-enclave visibility testing and declassifies an entity if found to be visible in a given frame (§6.4).

6.1 Bootstrapping

In this subsection, we go over the bootstrapping of BlackMirror, as an enclave module installed alongside the game client, as well as the establishment of a secure channel with the game server.

Installation of secure enclave. BlackMirror is deployed as an enclave, and which should be loaded by the game client. First of all, the user downloads the game client from the web. Installation of a game is done in a similar manner to typical games, except that the installation includes an enclave program.

Attestation of correctness. Later, the user runs the client asks a game server to join a game. Upon receiving a request, the server challenges the client to prove the correct loading of BlackMirror

on its platform. In particular we take advantage of remote attestation [13, 42] of SGX. As a result, the server can enforce that every connected client runs BlackMirror on an SGX-enabled CPU.

Establishing trusted channel with server. BlackMirror and the game server generate a shared secret key, using Elliptic Curve Diffie-Hellman (ECDH), that resides within the enclave and is valid for the duration of the game. The shared key is not exposed outside the enclave and the server will not perform this step without prior attestation. Furthermore, if the attacker stops running BlackMirror during gameplay, their program will terminate in an unstable state since server updates cannot be deciphered without the shared key. They will also lose all updates stored within the enclave memory. Lastly, server appends sequence numbers to the messages that it sends to the client, to prevent replay attacks.

6.2 Protecting Entities for Trusted Visibility Testing

The key idea of BlackMirror is to store potentially sensitive entities within an enclave. In order to still leverage GPU-based rendering, those secured entities are allowed to leave an enclave and pass to GPU only if known to be visible. Hence, BlackMirror needs trusted visibility testing such that it can faithfully tell entities can leave an enclave or not. In this regard, the following of this subsection describes which entities should be stored within an enclave for trusted visibility testing.

Entity Hierarchy and Visibility Testing. A hierarchy of game entities can be illustrated as shown in Figure 6. Entities can be classified into either self, sensitive, or non-sensitive entities. Self entity represents the player him/herself, from which the camera location is determined. Sensitive entities are the entities whose state should not be leaked, unless it is necessary, for example, the enemy entities or the supplies. On the contrary, the state of non-sensitive entities are not crucial for the game play, even if it is leaked, for example, the walls or environment objects. Non-sensitive entity can be classified into i) occluders, which may affect the visibility of sensitive entity (e.g., wall) and ii) non-occluders, which does not block the view.

Player's visibility is determined with following two factors: 1) a visible volume, which considers if an entity is within a field-of-view of the self-player; and 2) an occlusion, which considers if an entity is obstructed by other entities (such as wall). For example, in Figure 8, the visible volume is highlighted with shadow area, and e_2 and e_3 survives visible-volume-based clipping, as they are included in the shadow. However, e_1 is non-visible entity because it is placed outside the visible volume. Then considering the occlusion, e_2 becomes a non-visible entity as it is behind the wall, which is highlighted with darker shadow. As a result, only e_3 is the visible entity for the player, and e_1 and e_2 are non-visible entities.

Entities for Trusted Visibility Testing. BlackMirror protects the entities that are relevant to visibility testing, namely self, sensitive entities, and occluders. More specifically, BlackMirror needs to guarantee the confidentiality of sensitive entities, because enemy entity information itself is the target of game cheating. BlackMirror also needs to ensure the integrity of all of these entities, as they possibly affect the result of visibility testing. For instance, if an attacker is able to change the location of a box, which is an occluder,

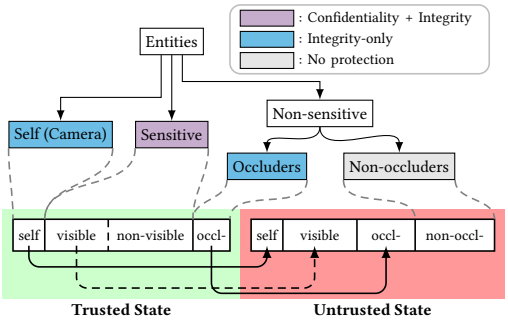


Figure 6: An entity hierarchy and BlackMirror's partitioning to trusted and untrusted state. Filled lines denote that entities are unconditionally declassified, and dashed lines denote that entities are declassified depending on the result of visibility testing.

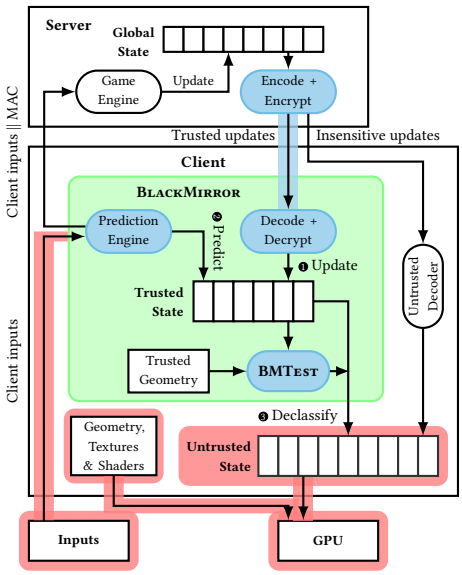


Figure 7: Overall architecture of BlackMirror

the attacker would be able to evade the visibility testing, so that an enemy entity which should not be visible, say it is behind the box, for the attacker can be tested as visible.

6.3 Trusted State Updates

BlackMirror stores sensitive entity states within an enclave as described in §6.2. Thus, BlackMirror should be able to update such entity states within an enclave so as to preserve the functionality of game plays. More specifically, there are two in-enclave updates to trusted state: (a) updates from the server through encrypted packets to ensure consistent game states; and (b) predictions performed by the enclave in response to input received from the player.

Updating Trusted State with Server Messages. The server periodically transmits encrypted packets, which includes changes to each entities including the player entity and sensitive entities.

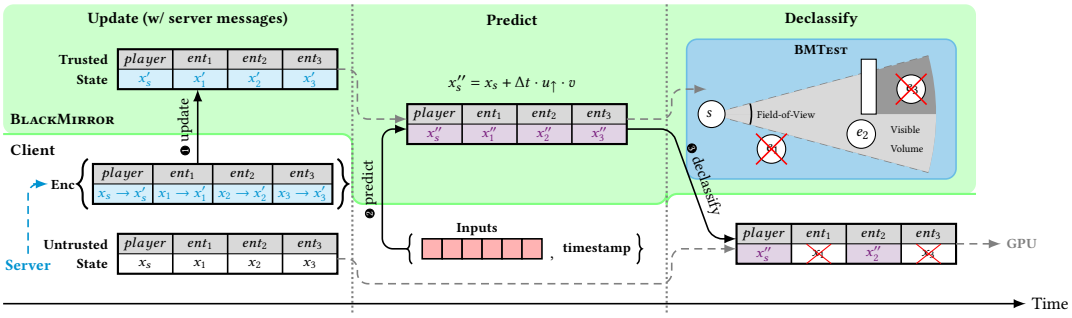


Figure 8: BlackMirror's workflow on update, predict and declassify operations

Upon receiving these packets, BlackMirror runs `update` operation (❶ in Figure 8) to synchronize its trusted state with the server. Updating the trusted state is done similarly to how the regular (insecure) client updates its local entity states (refer §2.1).

Local Prediction using Player Input. The client invokes `predict` operation (❷) to perform prediction-based state updates using player input. Recall that this local prediction is vital to ensuring that game scenes are updated more frequently (§2.1).

Calculating state updates using provided input. To perform predictions, BlackMirror requires player input, e.g., from keyboard and mouse, and the elapsed time from the last prediction to predict the future trusted state. To be more specific, it uses the input states, such as the fraction of time a key was pressed, and what was the mouse direction to compute final location of the player entity. For example, in Figure 8, the player's origin was x'_s after server updates. The time between the last prediction and current time is Δt and the player pressed forward (↑) key for given time interval. BlackMirror can compute unit vector of the forward direction using the view angle and call it be $u_\uparrow$. Then the predicted origin of the player becomes $x''_s = x_s + \Delta t \cdot u_\uparrow \cdot v$, where v is the speed value stored in the enclave.

Sensitive entities other than the player entities can be updated as well. In particular, BlackMirror extrapolates the origin and angle based on the elapsed time and previously received server messages. *Security Checks on Untrusted Player Inputs.* BlackMirror should also address malicious inputs during the prediction, since these inputs are received from untrusted devices. In particular, it ensures two security property of user-provided inputs: i) if the user-provided input matches the input sent to the server; ii) if the user-provided input violates the movement rule dictated by the game map. First, it latches raw inputs from the client (e.g., fraction of key-down time) for each invocation of prediction. Before the client forwards the input to the server, BlackMirror computes the MAC of the latched inputs, and send it along with the inputs. Therefore, the server-side simulation and local predictions are using the same inputs, which prevents discrepancy between the two, which may give attackers chances of fooling the BlackMirror to reveal more information. Second, BlackMirror checks if the user-provided input violates the movement rule in the game. To this end, BlackMirror performs simple collision detection in order to clip the predicted position against the environment, e.g., preventing the entities going through the walls during prediction.

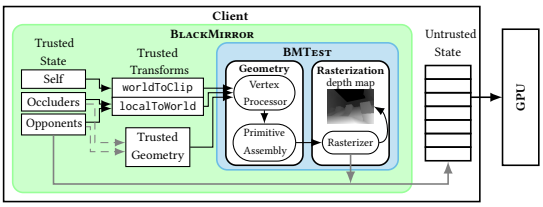


Figure 9: The workflow of BlackMirror's visibility testing

6.4 Trusted Visibility Testing

In this section, we describe how BlackMirror performs trusted visibility testing in an enclave with respect to those entities. Through trusted visibility testing, BlackMirror can securely determine which entities are visible in the current frame, and only declassify those entities outside the enclave, allowing them to be sent to the GPU. This declassification mechanism allows BlackMirror to still leverage GPU-based rendering pipelines, thereby overcoming the severe performance limitation of software-only rendering (§9.2).

More specifically, BlackMirror implements BMTest, a stripped-down version of graphics rendering pipeline, which is dedicated for visibility testing for a given game scene. Instead of implementing full-fledged rendering pipeline in software, BMTest only computes the necessary functions for visible volume-based clipping and depth-testing, and leaves the rest for the GPU to compute. In particular, BMTest performs vertex processing, primitive assembly and rasterization in software, (refer Figure 9) Additionally, BlackMirror can leverage SIMD instruction sets, e.g. AVX2, in the enclave to exploit parallel computations.

In the following, we explain the overall visibility-testing mechanism of BMTest including details about each step of its execution: i) preparation of transformations and rendering data, ii) Depth map construction with occluders, iii) Clipping by visible volume and iv) Depth-testing against the depth map.

Preparation of Transformations and Rendering Data. Initially, BlackMirror prepares inputs that are used by BMTest, i.e., transformations, and the geometry. Recall that we have discussed different coordinate systems (e.g., local space, world space, clip

space, etc.) in §2.2. Similar coordinate transformations are also performed within BMTest, yet these transformations are derived from trusted state. In particular, BlackMirror derives `worldToClip` transformation from the self-player's state, and it is used to determine the visible volume of the self-player at this frame. Then, it computes `localToWorld` transformations using the origin and view angles of occluders and sensitive entities.

Other than these transformations, BMTest requires the geometry (e.g., size and shape) of sensitive entities to perform accurate visibility testing. It is worth noting that since the geometry can be huge, BlackMirror stores only a simplified model of an entity's geometry, i.e., ignoring the details not relevant to determining shape and size. The reason for this is that EPC memory is limited and exhausting it can result in bad performance for enclaves [43, 44]. Furthermore, such a simplified model is commonly-used by game applications (e.g., for collision-detection) and should therefore, be trivial to implement for BlackMirror. We show the impact of different Level-of-Details (LoDs) on the accuracy of BMTest in §9.1.

After preparing the transforms and the rendering data (including trusted geometry), BlackMirror passes them to BMTest for performing visibility testing.

Depth Map Construction with Occluders. First, BMTest constructs a depth map, to accumulate depth-values of each occluder entity. The depth map later is used for depth-testing for the enemy entities. During this stage, BlackMirror iterates over the list of occluders in the trusted state, and passes the associated transformations and geometry to the BMTest to update the depth map.

For each occluder, BMTest transforms it into clip space, and then screen space to compute its depth values, and incrementally update the depth map. In particular, resolution of the depth map (i.e. the resolution of the screen space) is configured by the BlackMirror. We discuss the performance and security implications of the depth map resolution in §9.1.

In the following steps, enemy entities are handled according to the player's visible volume and recorded depth map values.

Clipping Sensitive Entities by Visible Volume. First, the geometry pipeline of BMTest discards enemy entities outside the visible volume. Similar to depth-map construction step, BlackMirror passes each sensitive entity in the trusted state, and its relevant rendering data to BMTest. After transforming the entity from the local space to the clip space with the geometry pipeline, BMTest can determine whether the enemy entity is outside the visible volume. The entities within the visible volume are passed to the rasterization pipeline for more precise depth testing.

Determining Visibility of Sensitive Entities. Finally, the rasterization pipeline tests each entity within the visible volume against the depth map. In particular, entities in the clip space are mapped to the screen space and BMTest compares the depth values of each entity against the values in the depth map. If the depth values of pixels in the entity are farther than the value in the depth map, it must be hidden behind some occluder, and therefore, should not be declassified. Lastly, BlackMirror declassifies the entities that are considered to be visible by the BMTest to the untrusted state. In particular, BlackMirror discloses their current trusted state to the untrusted state via shared memory, so that they can be rendered by the GPU.

Table 1: A list of possible attacks for a wallhack-motivated attacker and defenses provided by BlackMirror.

Attacks	BlackMirror Defenses
Accessing sensitive entities	
Compromising game client	Sensitive entities protected in BlackMirror (§6.2)
Malicious rendering requests	Trusted visibility testing for each frame (§6.4)
Using CPU-GPU channel Inside GPU memory	Only visible entities disclosed outside
Compromising visibility testing	
Tampering entity geometry	Trusted geometry stored in BlackMirror (§6.2)
Providing malicious input	Synchronize states with server (§6.3)
Undermining server communication	
Impersonate server	SGX attestation using public key (§6.1)
Replay attacks	Non-repeating sequences (§6.3)

7 SECURITY ANALYSIS

Table 1 provide an overview of possible attacks and we provide further details below.

7.1 Accessing Sensitive Entities

BlackMirror protects the identified sensitive entities from all attack surfaces that we have discussed in §3.2. In particular, given that the game client (running on the attacker's machine) can be easily compromised, BlackMirror identifies and then stores all sensitive game entities (and their corresponding states) in the enclave. Afterwards, these entities cannot be accessed by the attacker, until explicitly disclosed by `declassify` operation, preceded by BlackMirror's trusted visibility testing which filters out all entities not relevant to the current frame. As a result of filtering, the entity states are not disclosed outside the enclave, and therefore, attacks compromising CPU-GPU communication or the GPU memory are useless. In §9.1, we show that BMTest is highly accurate.

7.2 Compromising Visibility Testing

An attacker may attempt to tamper with inputs (i.e., entity geometry data and keyboard/mouse input) provided to BlackMirror to manipulate declassification of sensitive entities.

Tampering Entity Geometry. BlackMirror has to ensure the integrity of the geometry parameters of its sensitive entities to prevent manipulation of visibility testing results. For example, if the attacker can shrink the size of a wall, the opponents behind the wall will be classified as visible by BMTest. BlackMirror defeats such attacks by downloading geometry data via trusted channel with server during bootstrapping, and storing it within the enclave during game execution. Lastly, BlackMirror seals [13] the downloaded geometry data within the protected file system provide by SGX [45], to avoid offline tampering.

Providing Malicious Keyboard/Mouse Inputs. The attacker might try to manipulate BlackMirror's `predict` operation using malicious keyboard or mouse inputs from untrusted devices. As a result, they night be able to fool BMTest.

To prevent this, BlackMirror synchronizes sensitive entity states with the states sent to the game server through its secure communication channel. To understand why this is sufficient, consider that the attacker claims that it has moved beyond an occluder and therefore, should be able to see enemy characters beyond the occluder. While the attacker can lie to BlackMirror, if they do

Configuration	# Keyframes	# Vertices	Visualization
Bounding Box	198	8	
Precise	198	473	

Figure 10: The changes in the number of vertices depending on geometry's level of details (i.e., bounding box and precise)

not propagate this lie to the server, BlackMirror will be alerted to this discrepancy and terminate the game. On the other hand, if the attacker also lies to the game server, the server will update its global game state and propagate that state to all other players. As a result, the attacker's character will become visible to their opponents on their screens. Therefore, the end result of such a tampering will not be advantageous to the attacker beyond an honest gameplay.

7.3 Undermining Server Communication

The attacker might attempt to impersonate the game server or use replay attacks to undermine BlackMirror-server communication channel. To prevent impersonation, BlackMirror is provided with the server's public key and only establishes communication channels with the proper server after mutual authentication. To prevent replay attacks, the server should append non-repeating and increasing sequences to all update messages, and therefore, BlackMirror can verify that they are not replayed. It is worth mentioning that BlackMirror can adopt well established techniques [46] to address potential problems due to lossy UDP communications, for example, out-of-order arrival or packet losses.

8 IMPLEMENTATION

We develop a prototype of our design on top of an open source 3D FPS game, Quake II. While being quite old, Quake series have been studied by many previous research works [47–50]. Notably, NVIDIA has recently open sourced Q2RTX [51] to showcase real-time ray tracing, which is implemented atop Quake II. We argue that the choice of Quake II is quite reasonable for demonstrating our design. We choose Q2PRO [52] as our baseline, which comprises about 100K LoC. We use Intel SGX SDK version 2.7 for creating enclave, and software-based rendering functionalities, e.g., vertex processing and rasterization are taken from Masked Occlusion Culling library [53]. Additionally, we use AES128-GCM and AES-CMAC for encryption/decryption and MAC, respectively, which are available from SGX SDK and OpenSSL.

9 EVALUATION

Experimental Setup. We run both of BlackMirror-protected Quake II server and client on a desktop machine with Intel i7-8700 (6 core), 16 GB RAM, NVIDIA GeForce RTX 2080 Ti with 11GB GDDR6. Both run Ubuntu 18.04. We intentionally restrict BlackMirror to a single thread, expecting most CPU cores are occupied by the rest of the game client, namely networking, handling inputs, and rendering. [34, 54]

9.1 Accuracy and Overhead of BMTest

In this section, we evaluate how accurately and efficiently BMTest can perform visibility testing. We analyze the accuracy of visibility testing with a publicly available demo [1] with different configurations. In particular, we experiment with i) two different Level of Defails (LoDs); bounding boxes and precise models shown in Figure 10, and ii) different enclave depth map resolutions: 360p, 720p and 1080p to evaluate their impact on security and performance.

Accuracy of Visibility Testing. BlackMirror can be configured to have different geometry's level of details (LODs), which may affect the accuracy of declassification. In the following, we measure the accuracy and the false negative rate of BMTest with two different LODs, namely with bounding boxes and with precise models. As shown in Figure 10, both bounding box and precise model have the same 198 key frames while having 8 and 473 vertices, respectively. Note that the precise model has 59 times more vertices enabling the detailed rendering compared to the bounding box.

To evaluate the accuracy of BlackMirror's visibility testing, we replay the demo and measured (A) the total number of rendering attempts for enemy entities in all the frames (i.e., the number of enemy entities, including the visible and invisible ones), (B) the number of visible entities that passes BMTest with the bounding box and precise model, respectively, (C) the number of entities that are not declassified by BMTest, though it should have, and (D) the true number of visible entities (the ground truth). Note that to measure D, we leverage the occlusion query extension [55], which allows a developers to count that number of triangles that are drawn without being occluded, thus providing the true number of entities that are rendered to the final scene. The accuracy and the false negative rate of visibility testing is defined to be $1 - \frac{D-B}{A}$ and $\frac{C}{A}$, respectively.

The results are shown in Figure 11. Overall the accuracy of BlackMirror's visibility testing ranges from 0.973 to 0.986, implicating that BlackMirror effectively filters out non-visible entities. On the other hand, there exist few false negatives, due to low resolution of enclave depth map, and different algorithms being used by CPU and GPU. The false negative rate turns out to be small, and we expect it can be further reduced by combining techniques such as path-tracing, i.e., casting a ray from the camera to the entity, nevertheless we leave specific implementations to be future works.

Performance Impact of Visibility Testing. The performance of BlackMirror's visibility testing depends on: 1) what is the resolution of the depth map? and 2) how detailed the provided geometry is?

Figure 12 shows the performance overhead of the depth map preparation. Depending on the resolution, for each frame it takes from 0.36 millisecond (for 360p resolution) to 0.50 millisecond (for 1080p resolution). Since most games require 60fps, the time gap between frames is about 16 millisecond. In this regard, the overhead of depth map construction (i.e., 0.36 to 0.50 millisecond) only takes about 0.02% to 0.03%, we believe such overhead should be acceptable to meet gaming requirements for user's experience.

Figure 13 shows a comparison of the depth testing performance while varying i) the number of sensitive entities to test and ii)

[1] http://acmectf.com/downloads/demos-tricks/_unsorted/challenge-tv/ffadm1.dm2.zip

Enclave Depth Map Resolution	Total # of Frames	Total # of Entities (A)	# of Visible Entities w/ Bbox (B) ($1 - \frac{D-B}{A}$)	# of Visible Entities w/ Precise (B) ($1 - \frac{D-B}{A}$)	# of False Negative w/ Bbox (C) ($\frac{C}{A}$)	# of False Negative w/ Precise (C) ($\frac{C}{A}$)	Ground Truth # of Visible Entities (D)
360p	299,832	708,907	182,944 (97.3%)	176,686 (98.2%)	13 (0.0018%)	262 (0.037%)	163,834
720p	295,044	693,976	176,832 (97.6%)	170,606 (98.5%)	12 (0.0017%)	216 (0.030%)	160,009
1080p	283,293	658,824	166,007 (97.8%)	160,560 (98.6%)	10 (0.0015%)	185 (0.026%)	151,185

Figure 11: Accuracy and false negative rate of BlackMirror's visibility testing

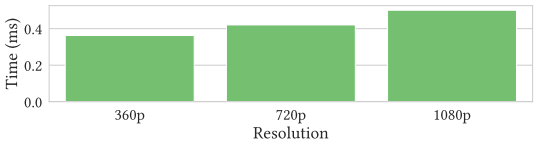


Figure 12: The time taken to prepare a depth map (per frame) while varying its resolution

testing depth resolution. Overall, for all different cases we believe the visibility testing time is negligible (all cases are less than 0.50 millisecond), suggesting very little impact on user experience. Furthermore, the precise geometry takes more time than the bounding box geometry, as expected since it has more vertices to check. Similarly, as the number of sensitive entities increases, the visibility testing takes more time because it has to enumerate all of them.

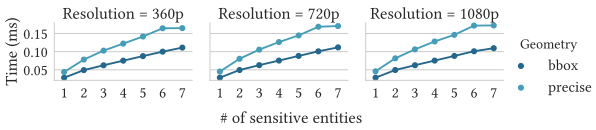


Figure 13: A comparison of the depth testing performance while varying i) the number of sensitive entities to test and ii) testing resolution. The time is measured with a per-frame depth testing time.

9.2 End-to-End Performance Evaluation

To understand end-to-end performance impact of BlackMirror, we run our fully-functional BlackMirror-protected Quake II with two participants. In particular, we evaluate if client and server overhead are acceptable to determine if BlackMirror has low impact on client experience and is scalable for the server to implement.

Client-side Overhead. We partition the client application into three parts: update, predict and render, and measure average execution time of these periods, with three different settings. In particular, the client updates the game states with server messages during the update period, and it predicts the player state using local inputs during the predict period. Then, for the render period, the client render the scene on the GPU or with the software renderer. Note that for BlackMirror, update and predict take places in the enclave and it also performs trusted visibility testing in advance to rendering on the GPU, which are the source of the overhead presented by BlackMirror. For the baseline, we run unmodified game client, which runs non-enclave update and prediction, and renders the

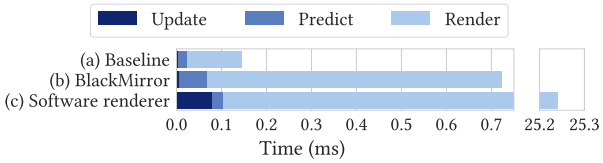


Figure 14: A comparison of the overhead to run a frame. (a) Baseline runs updates and predictions without an enclave, and renders the scene on the GPU. (b) BlackMirror runs updates and predictions within the enclave, and performs trusted visibility testing before rendering the scene on the GPU. (c) Software renderer is identical to the baseline, except that it renders the scene with a software renderer.

scene on the GPU (a). BlackMirror runs update and prediction within the enclave, and performs trusted visibility testing before rendering the scene on the GPU (b). Lastly, we run non-enclave client with a software renderer, SwiftShader [37] (utilizing 12 threads), to give the sense of performance overhead incurred by adopting one of the straw man designs discussed in §5 (c). (See **D2**)

The result is shown on Figure 14, and overall, the evaluation result show that that BlackMirror adds in total 0.57 ms overhead per each frame on average. Therefore, if the native game operates at 60 frames-per-second, i.e., takes 16ms to render each scene (common setting for modern games), BlackMirror demonstrates 58-59 frames-per-second on average, which is negligible and acceptable. The performance overhead of BlackMirror is mainly caused by running updates and predictions within the enclave, and performing the trusted visibility testing before rendering on the GPU. On the other hand, software-rendering with SwiftShader [37] exhibits 34× slowdown than BlackMirror, even when running without enclave and utilizing 12 threads.

Server-side Overhead. Since BlackMirror-based Quake II requires secure channel to communicate, server performs encryption over all packets sent to and received from clients, unlike native Quake II. Figure 15 measures the time taken to encrypt a packet while varying the size of packets. As shown in the figure, the encryption overhead is always less than 0.23 ms (0.06 ms on average), which should not interfere the gaming experience for 60 fps games.

10 DISCUSSIONS

Applicability to a broader range of games. Although we prototype BlackMirror on Quake II, we expect that our approaches be applicable to a broader range of games for two main reasons: (i) its interface is general enough to be adopted by most multiplayer

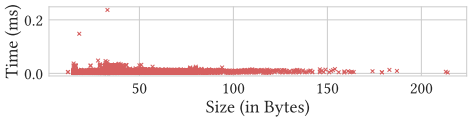


Figure 15: The overhead of packet encryption at server

shooter games, and (ii) modern game engines are equipped with features, which can be retrofitted to realize our design.

First, we investigate the client applications of latest open-source 3D multiplayer shooter games, Xonotic [56] and Red Eclipse [57] to show that they also share the same game architecture (refer to §2) with Quake II. In particular, within the main processing loop, all of them performs the following operations: (a) updating the game state with server messages, (b) predict the player state with inputs (optionally run a physics engine), and (c) render the scene with updated view values. From the observation, we conclude the interfaces of BlackMirror can be smoothly integrated to these games without intrusive changes to their architecture. Furthermore, we expect closed-source games also follow the same architectural footprint; therefore, BlackMirror can be applied to them, as well.

Second, we find that modern game engines including Unreal Engine already provide in-CPU visibility testing features [58, 59], one of the core requirements for BlackMirror, therefore the game engine developers can easily adopt our approaches. However, note that their goal for testing visibility on CPU is to improve the performance rather than preventing wallhacks.

Advanced Rendering Techniques. Although we successfully demonstrate a prototype of BlackMirror using nontrivial game, it may require further efforts to be integrated to games with more complex rendering pipelines. Firstly, modern game engines often transform the shape of entities on GPU, e.g., geometry and tessellation shaders. In order to apply BlackMirror to games, the game developers have to use conservative trusted geometry, so that it can tolerate possible updates to the shapes on the GPU. Secondly, an entity behind the wall may affects the scene with complex graphics engines. For example, the shadow of a hidden entity may appear to the scene, or it may be a light source, so that the neighboring pixels are enlightened. We can modify BMTest take account of these effects when making decisions on the visibility, but we leave specific implementations as future works.

11 LIMITATIONS

Aimbots. Aimbots automatically generates inputs (e.g., moving mouses) to move cursors on the enemy at will, so that the attacker can make more accurate shots. BlackMirror does not prevent these attacks since it does not protect visible enemy states that can be used to calculate required fake inputs, and without trusted input devices [15, 16], it is hard to distinguish between artificial inputs from the genuine user inputs. That said, current version of BlackMirror will not obsolete the entire existing anti-cheat techniques due to lacking support for other types of cheats, namely the aimbots. Nevertheless, we believe BlackMirror is an important step towards bringing TEE technologies to cheat prevention in online game, and therefore eliminating over-privileged current anti-cheat software.

Noticeability vs. visibility. In some game, the vision of a player is hindered by environmental objects (e.g., bush or foliage) and particle effects (e.g., explosion), which partly covers sensitive entities, or has similar colors with them. Attackers may attempt to nullify these features by leaking their position (as far as the entities are only partly covered) or modifying the textures so that the entities can be easily noticed. It is hard for BlackMirror to prevent these attacks, since the state of the entities would be declassified, unless the entire object is hidden behind the wall, and due to lack of trusted memory region on the GPU (for tamper-evident textures).

12 RELATED WORKS

OpenConflict [34] invents an multi-party computation (MPC) protocol for preventing maphacks in 2D RTS games. In OpenConflict, each client computes its current visible area, and it is obliviously sent as a query to its opponents who return a list of their units that are overlapping with requested visible area. However, such MPC schemes are not suitable for 3D games, where testing visibility is much more complex. In particular, in 3D world, the visible area is typically larger than the 2D world (imagine visible area spanning to almost infinity in 3D games, but cut by a certain distance in 2D games), thus the query should be very large. Additionally, the view is obstructed by various objects, which further complicates computing visible volume itself in advance to MPC protocol. Therefore, such an MPC protocol will likely be inefficient in 3D games. AVM [60] provides accountability to the execution of virtual machine by tamper-evident logging and deterministic replay. Watchmen [49] uses distributed proxies for cheat prevention in multi-player games. Watchmen focuses more on distributed proxy architecture, instead of specific visibility testing mechanisms, e.g., how to precisely compute a vision set. Bauman et al. [61] showcased how to leverage Intel SGX technology for protecting games. However their work mainly focus on DRM, while leaving solutions for cheat prevention as a future work.

13 CONCLUSION

Online game cheating, particularly wallhacks, is a critical issue for a competitive game, and anti-cheat solutions by far are fundamentally flawed because it is deployed on a machine where the attacker has absolute control. This paper presented BlackMirror, a new game design with a trusted execution environment, Intel SGX. It leverages strong data isolation guarantees provided by SGX to prevent wallhacks. The implementation and evaluation with BlackMirror-protected Quake II demonstrate that BlackMirror can enable fully functional and secure games while meeting user experience requirements on games.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Yan Shoshitaishvili, for the insightful and thoughtful feedback which guided the final version of this paper. This work was partly supported by National Research Foundation (NRF) of Korea grant funded by the Korean government MSIT (NRF-2019R1C1C1006095). The Institute of Engineering Research at Seoul National University provided research facilities for this work.

REFERENCES

[1] Global games market report. https://newzoo.com/products/reports/global-games-market-report/.

[2] Report: Cheating is becoming a big problem in online gaming. https://www.forbes.com/sites/nelsongranados/2018/04/30/report-cheating-is-becoming-a-big-problem-in-online-gaming.

[3] Battleye: The anti-cheat gold standard. https://www.battleye.com. Accessed: 2020-01-03.

[4] Valve anti-cheat system (vac). https://support.steampowered.com/kb/7849-RADZ-6869/#whatisvac. Accessed: 2020-01-03.

[5] Easy anti-cheat. https://www.easy.ac/en-us/. Accessed: 2020-01-03.

[6] Joel Noguera. Unveiling the underground world of anti-cheats. Recon Montreal 2019, 2019.

[7] Cheat engine. https://www.cheatengine.org/. Accessed: 2020-01-02.

[8] Joel Noguera. Creating your own wallhack. https://niemand.com.ar/2019/01/13/creating-your-own-wallhack/, January 2019.

[9] Carl Schou. Hooking the graphics kernel subsystem. https://secret.club/2019/10/18/kernel_gdi_hook.html, October 2019.

[10] Peter Andreasen. Deep dive into networking for unity's fps sample game. Unite LA, 2018.

[11] Quake 2 gpl release. https://github.com/id-Software/Quake-2.

[12] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[13] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for cpu based attestation and sealing.

[14] Samuel Weiser and Mario Werner. Sgxio: Generic trusted i/o path for intel sgx. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, 2017.

[15] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. Garcia, E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh. Fidelius: Protecting user secrets from compromised browsers. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[16] Aritra Dhar, Enis Ulqinaku, Kari Kostiainen, and Srdjan Capkun. Protection Root-of-trust for io in compromised platforms. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2020.

[17] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, October 2018.

[18] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity gpus. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, April 2019.

[19] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, San Jose, CA, May 2015.

[20] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, August 2017.

[21] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.

[22] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, SysTEX'17, 2017.

[23] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltomore, MD, August 2018.

[24] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[25] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltomore, MD, August 2018.

[26] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, August 2017.

[27] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel SGX. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltomore, MD, August 2018.

[28] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[29] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17. ACM, 2017.

[30] Nicilas Guigo and Joel St. John. Next level cheating and leveling up mitigations. Black Hat Europe 2014, 2014.

[31] Osiris. https://github.com/danielkrupinski/Osiris.

[32] Charlatano. https://github.com/Jire/Charlatano.

[33] Onebytewallhack. https://github.com/danielkrupinski/OneByteWallhack.

[34] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh. Openconflict: Preventing real time map hacks in online games. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2011.

[35] imgui. https://github.com/ocornut/imgui.

[36] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[37] Swiftshader. https://github.com/google/swiftshader.

[38] Xigncode3. https://www.wellbia.com/home/en/pages/xigncode3/. Accessed: 2020-01-03.

[39] Riot's approach to anti-cheat. https://technology.riotgames.com/news/riots-approach-anti-cheat. Accessed: 2019-12-28.

[40] Carl Schou. Battleye anticheat: analysis and mitigation. https://vmcall.github.io/reversal/2019/02/10/battleye-anticheat.html/, February 2019.

[41] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzeland, April 2020.

[42] Guoxing Chen, Yinqian Zhang, and Ten-Hwang Lai. Opera: Open remote attestation for intel's secure enclave. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2018.

[43] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A data oblivious filesystem for intel SGX. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[44] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2017.

[45] Surenthar Selvaraj. Overview of protected file system library using software guard extensions, 2016.

[46] Timothy Ford. Overwatch gameplay architecture and netcode. GDC 2017, 2017.

[47] Daniel Lupei, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, Paris, France, April 2010.

[48] Vladimir Gajinov, Ferad Zyulkyarov, Osman S. Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. Quaketm: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd International Conference on Supercomputing (ICS)*, Yorktown Heights, NY, June 2009.

[49] A. Yahyavi, K. Huguenin, J. Gascon-Samson, J. Kienzle, and B. Kemme. Watchmen: Scalable cheat-resistant support for distributed multi-player online games. In *Proceedings of the 33st International Conference on Distributed Computing Systems (ICDCS)*, 2013.

[50] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Releigh, USA, February 2009.

[51] `Q2RTX`. https://github.com/NVIDIA/Q2RTX.

[52] `Q2PRO`. https://github.com/skullernet/q2pro.

[53] Masked software occlusion culling. https://github.com/gametechdev/maskedocclusionculling.

[54] J. Hasselgren, M. Andersson, and T. Akenine-Möller. Masked software occlusion culling. In *Proceedings of High Performance Graphics*, HPG '16, 2016.

[55] ARB_occlusion_query. https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_occlusion_query.txt.
[56] Xonotic. https://github.com/xonotic/xonotic.
[57] Red eclipse 2. https://github.com/redeclipse/base.
[58] Visibilty and occlusion culling. https://docs.unrealengine.com/en-US/Engine/Rendering/VisibilityCulling/index.html.
[59] Michal Valient. Practical occlusion culling in killzone 3: Will vale — second intention limited — contract r&d for guerrilla bv. In *ACM SIGGRAPH 2011 Talks*, SIGGRAPH '11, 2011.
[60] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.
[61] Erick Bauman and Zhiqiang Lin. A case for protecting computer games with sgx. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, SysTEX '16, 2016.