Cheating in networked computer games – A review

Steven Daniel Webb and Sieteng Soh

Curtin University of Technology
Department of Computing
Perth, Western Australia
+61 8 9266 7680

{steven.webb@postgrad,soh@cs}.curtin.edu.au

ABSTRACT

The increasing popularity of Massively Multiplayer Online Games (MMOG) – games involving thousands of players participating simultaneously in a single virtual world - has highlighted the scalability bottlenecks present in centralised Client/Server (C/S) architectures. Researchers are proposing Peer-to-Peer (P2P) architectures as a scalable alternative to C/S; however, P2P is more vulnerable to cheating as it decentralises the game state and logic to un-trusted peer machines, rather than using trusted centralised servers. Cheating is a major concern for online games, as a minority of cheaters can potentially ruin the game for all players. In this paper we present a review and classification of known cheats, and provide real-world examples where possible. Further, we discuss counter measures used by C/S architectures to prevent cheating. Finally, we discuss several P2P architectures designed to prevent cheating, highlighting their strengths and weaknesses.

Categories and Subject Descriptors

C.2.0 [**Computer-communication networks**]: General - *Security and protection*.

General Terms

Algorithms, Security, Human Factors, Theory, Verification.

Keywords

Cheating, client/server, networked computer games, peer-to-peer.

1. INTRODUCTION

Massively Multiplayer Online Games (MMOGS) differ from traditional network games as they present a single universe in which thousands or tens of thousands of players interact simultaneously [16]. Further, the state of the game world and player's avatars progresses gradually, lasting months or years. In the last five years the popularity of MMOGs has increased dramatically; enabled by the explosive growth of the Internet and the availability of broadband connections for home users.

DIMEA'07, September 19–21, 2007, Perth, WA, Australia.

Most networked computer games use a Client/Server (C/S) architecture where all players connect into a central trusted server that simulates and validates the game. To support that massive number of concurrent players, the vast majority of commercial MMOG available today use multiple co-located servers to distribute the processing requirements of simulating the world. C/S architectures are ubiquitous for MMOG as they have the following benefits. Firstly, they have proven successful, an important factor when approaching venture capitalists for funding. Secondly, they are secure against most forms of cheating. Thirdly, centralised architectures give the publisher strong control over the game. Finally, developing centralized architectures is easier than distributed architectures as the communications model is simple. Unfortunately C/S architectures have poor scalability, as the server(s) are often a processing and/or bandwidth bottleneck [1,8,14,15,16,23,33].

The scalability of an architecture is its ability to support a large number of concurrent players, and tolerate a rapid increase in the number of players without dramatically increasing the usage of centralised resources. To prevent the servers becoming a bottleneck, publishers must provision large amounts of hardware and bandwidth [1,8,14,15,16,23,33]. Furthermore, as more resources cannot be deployed rapidly, publishers often over-provision resources, to allow for a rapid grow in the number of players [6]. The cost of provisioning sufficient resources often prevents small companies from developing MMOG, and even some large developers struggle to provision sufficient resources [17,31].

The most common approach to solve scalability issues is *sharding* [4]. A shard is a complete and independent copy of the game world. The maximum number of concurrent players in a shard is bounded. By adding more shards the developer can accommodate more players; however, players in different shards cannot interact, thus sharding works against the concept of MMOG. Furthermore, it is frustrating and annoying for players when shards reach their limits and they must play on different shards with different people, destroying the social aspect of MMOG.

World of Warcraft (WoW) is arguably the most popular MMOG to-date, with over 8.5 million players worldwide. The WoW universe is sharded into many mutually exclusive worlds. Each shard is limited to several thousand concurrent players. Despite the massive success of WoW and the huge revenue it is generating, WoW has been plagued with scalability issues [31]. Shards rapidly reach their player limits, resulting in long queues of players waiting to join the shard. Several quests that result in a

large number of players generating a large number of events have been known to crash the server.

In recent years Peer-to-Peer (P2P) overlay networks have become an active research topic [29], in which peers exchange information directly, without routing it through a central server. The primary advantage of P2P over C/S is its scalability [1,8,14,15,16,23,33]. For every node that joins the system and makes requests, the node also provides resources to the system to handle the requests of other nodes; hence, P2P architectures are resource growing. There are real-world P2P systems that can scale up to millions of concurrent users [29]. While P2P architectures can potentially solve the bandwidth and processing power bottlenecks of centralized architectures they are more vulnerable to cheating.

Cheating is a major concern in network games as it degrades the experience of the majority of players who are honest [22]. This is catastrophic for games using subscription models to generate revenue [12]. To be a viable alternative P2P architectures must prevent cheating. We define cheating as *a user action that gives an advantage over his/her opponents that is considered unfair by the game developer*. Note, cheat prevention is a subset of security issues for online games.

In this paper we present a review and classification of known cheats, and provide real-world examples where possible. Further, we discuss counter measures used by C/S architectures to prevent cheating. Finally, we discuss several P2P architectures designed to prevent cheating, highlighting their strengths and weaknesses.

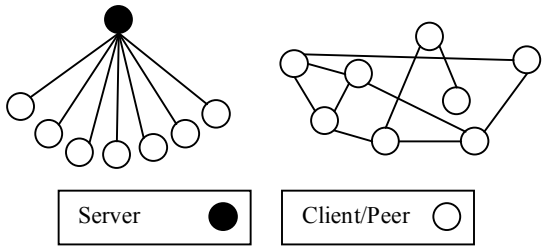
The layout of this paper is as follows. Section 2 provides a background into the strengths and weaknesses of C/S and P2P architectures. Section 3 discusses related reviews and classifications of cheating. Section 4 describes each form of cheating, gives real-world examples, and discusses possible countermeasures for C/S and P2P architectures. Section 5 reviews P2P architectures specifically designed to prevent cheating, and discusses their strengths and weaknesses. Finally, Section 6 concludes the paper. Note, "he" should be read as "he or she" throughout this paper.

2. CLIENT/SERVER AND PEER-TO-PEER

As shown in Figure 1(a), all players/clients in Client/Server (C/S) architectures connect to centralized servers. The servers are centralized trusted authorities whose tasks include: T1 - receiving player updates, T2 - simulating game play, T3 - validating and resolving conflicts in the simulation, T4 - disseminating updates to clients, T5 - storing the current game state, T6 - storing the offline player's avatar state, and T7 - authenticating players, downloading their avatar state, and billing. Task T3 is particularly important when preventing cheating as the server is trusted to fairly simulate game play and detect cheating. T7 is an important task for publishers, as billing is a vital part of most MMOG revenue models. Finally, the networking code for C/S application is very simple making the game easy to develop, and avoids issues related to firewalls.

While C/S is simple, secure, and reliable, it has limited scalability to support a large number of players as the servers often become a bandwidth and processing bottleneck [1,8,14,15,16,17,23,33], requiring developers to provision large amounts of hardware and bandwidth. Both inbound and outbound bandwidth may cause a bottleneck as the publisher must provision sufficient bandwidth for T1 and T4 at one location, which is an expensive re-occurring cost [22]; however, outgoing bandwidth is usually more critical as a single update from a client is often forwarded on to multiple other players. The server's processing power is another potential bottleneck, as it must handle T2 and T3, as well as calculating player's AoI in T4. C/S architectures also give an unfair advantage to players geographically close to the server, as they will have lower game delay (response time) than those situated further away [4]. Furthermore, redirecting updates through the server (T1 to T4) increases delay while consuming bandwidth and processing power. Finally, poor design may result in a server being a single point of failure for the system.

The Mirrored Server (MS) architecture is an extension to C/S that increases scalability by using multiple mirrors at different locations connected via a well provisioned network – low delay, high bandwidth, multicast enabled. By distributing the mirrors across multiple locations the bandwidth does not need to be provisioned at a single location, reducing cost and increasing scalability; however, the synchronization algorithm used increases the processing bottleneck present in C/S.



(a) Client/Server (C/S) (b) Peer-to-Peer (P2P)

Figure 1. C/S and P2P architectures.

Figure 1(b) depicts a typical P2P architecture in which there are no servers, and all updates are passed directly between players. Peers typically connect to other peers who posses relevant information, forming a structured set of links on top of the network (a network overlay). P2P architectures may utilize servers; however, they are characterized by peers (player machines) exchanging updates, rather than routing them through a server. P2P has the following benefits over C/S: (i) P2P architectures are resource growing (the bandwidth and processing capacity increases with the number of players) making them very scalable; (ii) delay is minimised as updates take a direct route between players; and (iii), there is no single point of failure. The scalability of P2P can potentially alleviate the high start-up cost of provisioning large amounts of bandwidth and processing power [22]; therefore, allowing small developers to create MMOG. However, maintaining security in P2P architectures is difficult as the simulation and game state is distributed to un-trusted peer machines; consistency algorithms are required to prevent errors in player's game state; and redundancy must be built in as individual player machines have a high probability of failure. Further, subscription based P2P network games will require a gateway server for player authentication and billing. Several hybrid C/S and P2P architectures that combine the security advantage in C/S and scalability in P2P have also been proposed [25,33].

3. CHEAT CLASSIFICATION

The first review of cheating and cheat detection/prevention was done by Matt Pritchard [26], one of the developers *of Age of Empires*, and presents his experiences both as a developer and player of computer games. This industry focused article discusses specific real-world cheats, and covers practical methods to

discourage them. Pritchard acknowledges that many of the solutions presented do not prevent cheating, but make it far more difficult for players to cheat. He argues that if the difficulty of cheating is greater than the difficulty involved in playing the game players will not cheat.

Following this industry focused article, Yan [35] provides a theoretical review of cheats, particularly focused on online Bridge. Of particular interest is his discussion of preventing collusion between players. While there are several possible counter-measures, for all practical purposes it is impossible to completely eradicate collusion in online games.

Kabus *et al.* [15] discuss three different techniques that may be used in P2P architectures to prevent/detect cheating: mutual checking, log auditing, and trusted computing. The principle of mutual checking is that *you may not trust a single client, but you trust the consensus of multiple unaffiliated clients*; therefore, multiple randomly selected – and hence trusted – clients are used to validate player actions before the game state is modified, preventing cheating. The second approach, log auditing, does not prevent cheating, but allows the game to detect cheating when it has occurred – albeit much later in some games. When cheating is detected the game performs a rollback to undo the effects of the cheat. Log auditing is not appropriate for all forms of games, particularly MMOG that do not have an *end state* (game completion). The final solution, trusted computing, involves using special hardware that prevents cheaters from modifying the game or running cheating programs. This solution is currently inappropriate for PC users; however, it is being actively used in console games. Unfortunately, several trusted computing solutions have been shown to contain weaknesses allowing players to run un-trusted programs, allowing them to cheat.

Yan and Randell [36] provide an extensive list of cheating techniques, and formed a taxonomy with regard to the underlying vulnerability (*what is exploited*?), consequence (*what type of failure can be achieved*?) and the cheating principle (*who is cheating*?). The authors find that traditional methods of security in software - confidentiality, integrity, availability, and authenticity - are necessary but insufficient to defend against cheating. Although large and detailed in its taxonomy, their characterization of cheating lacks structure, and it is argued that new forms of cheating cannot be easily integrated [23]. Note, we do not include several categories of cheating proposed by Yan and Randell (cheating by denying service to peer players, cheating by compromising passwords, cheating by exploiting lack of authentication, cheating by compromising game server, cheating related to internal misuse, and cheating by social engineering [36]) in our classification as they are relevant to all secure network applications and not directly related to game mechanics; hence, we believe they are general security issues, not cheating issues.

Neumann *et al*. [23] distinguish three categories of cheating based on the threatened game property: confidentiality, integrity, and availability. Confidentiality requires that a cheater cannot access state information they are not entitled to (secret information), else they will have an unfair advantage when selecting what actions to take. Integrity ensures that a cheater cannot modify the game state unfairly; hence, the integrity of the game state is maintained. Availability requires that the entire game is available to all players at all times, and that cheaters cannot prevent the game from progressing fairly. Unfortunately, this paper only provides a brief coverage of cheating; hence, the authors only briefly discuss possible cheats and their methods of attack.

GautheierDickey *et. al.* [14] proposed a cheat classification scheme comprising four categories: game, application, protocol, and network. Game cheats do not require any external programs or modification and occur entirely within the game; application cheats require using or modifying applications; protocol cheats interfere with the game's communication protocol; and network cheats involve modifying the network infrastructure over which the game traffic is sent. The authors only consider nine forms of cheating: denial of service, fixed delay, timestamp, suppressed update, inconsistency, collusion, secret revealing (information exposure), bots/reflex enhancers, breaking game rules, and their classification scheme is slightly too narrow to classify new forms of cheating.

Webb, *et al*.[33] modified the classifications scheme in [14] to be: game, application, network, and infrastructure. Infrastructure cheats involve modifying or manipulating pieces of infrastructure that the game relies on, such as drivers, libraries, hardware, the network, *etc*. Further, we classified 15 cheats using this scheme. In [33], we also proposed a hybrid C/S and P2P architecture that provides security equal to that in C/S. See Section 5.5 for details.

In this paper we extend the known cheats in [33] by including Real Money Transactions (RMT) (shown in Table 1). Furthermore, we include references to real-world examples of where these cheats have been used.

Table 1. Game cheats and their possible solutions

Cheat	C/S	PB/VAC2	AS	NEO/SEA	RACS
Game Level					
Bug	✓	×	✓	✓	✓
RMT	✓	×	×	×	✓
Application Level					
Information Exposure, Invalid Commands	✓	×	×	×	✓
Bots/reflex enhancers	×	✓	×	×	×
Protocol Level					
Suppressed update, Timestamp Fixed delay, Inconsistency	✓	×	✓	✓	✓
Collusion	☒	☒	☒	☒	☒
Spoofing, Replay	✓	×	×	✓	✓
Undo	N/A	×	✓	×	N/A
Blind Opponent	N/A	×	N/A	N/A	✓
Infrastructure Level					
Information Exposure	✓	✓	×	×	✓
Proxy/Reflex Enhancers	☒	☒	☒	☒	☒

4. CHEATS, EXAMPLES, AND COUNTERMEASURES

Table 1 classifies cheats into four levels: *game, application, protocol*, and *infrastructure*; some cheats fall into multiple levels (*e.g.*, Information Exposure). A ✓ indicates the cheat is solvable using the corresponding technology; an × if it is not solved; N/A

indicates the cheat is not applicable; and the rows of ☒ indicate that collusion, and proxy/reflex enhancers are not solvable. Note, PunkBuster (BP) [27] and Valve Anti-Cheat (VAC2) [32] can be combined with any of the other technologies.

4.1 Game level cheats

Game level cheats occur completely within the game program without any modification or external influence. In the following we describe two game level cheats: *bugs* and *real money transactions*.

4.1.1 Bugs

Bug cheats exploit design or implementation errors to gain an unfair advantage. Bugs do not require an in-depth knowledge/understanding of how/why they work, and do not require any additional programs or modifications to use. For example, the player ranking system in Warcraft II contained a designed error that gave rise to the win trading cheat; in which colluding players will repeatedly start matches against each other and then alternately surrender to give each of them the opponent victory points; thus, cheaters can climb to the top positions of the ranking ladder without playing any valid matches [35]. In Warcraft III win trading is prevented by randomizing the participants when creating matches; however, this only works if the pool of players is large. Another solution involves including losses as well as victories into the ranking function. An example of an implementation bug occurred in Halflife, where a specific combination of actions allowed cheaters to re-load weapons faster than honest players [26]; a significant advantage in First Person Shooters (FPS). This cheat was fixed by a software patch from the developer, which is the accepted method of preventing bug cheats. For MMOG a database rollback to an earlier state may be required if the cheat seriously influenced the game world. As bugs are present in both C/S and P2P architectures neither is resistant to this form of cheating. The accepted solution amongst the gaming industry and players for both C/S and P2P architectures is to release game patches to prevent the cheat; however, [12] argue that runtime verification may be used to prevent bug cheats.

4.1.2 Real Money Transactions (RMT)

A Real Money Transaction (RMT) is when a player purchases a game item or virtual currency using a real-world currency [18]. Many Asian MMOG use RMT as their revenue model (free to play, but it costs money to purchase items); however, most (but not all) western MMOG explicitly forbid RMT in their *End User License Agreement* (EULA), and the practice is considered cheating. Gold farming is a related phenomenon where low paid workers – usually in China – work full time playing MMOG to earn valuable items which are sold to players [18]. RMT occur outside of the game, often on auction sites such as e-bay, or on dedicated message boards [30]. Most MMOG suffer from RMT; however, WoW is one of the most highly targeted games by gold farmers due to its massive player base. WoW's publisher Blizzard regularly bans hundreds of thousands of players for gold farming in WoW [5]. While the method used by Blizzard to detect gold farmers is unknown, we suspect they use statistical analysis of log files generated by the servers. As most P2P architectures distribute the game state and logic to peers, retrieving the required information to perform statistical analysis is far more difficult or impossible in P2P than in C/S.

4.2 Application level cheats

Application level cheats require either modifying the game executable or data files, or running programs that read from/write to the game's memory while it is running. Developing application level cheats requires knowledge about reverse engineering; however, using them is trivial. In the following, we describe three application level cheats: *information exposure*, *bots/reflex enhancers*, and *invalid commands*.

4.2.1 Information Exposure

Also called secret revealing, the information exposure cheat results in the cheaters gaining access to information that they are not entitled to, such as their opponent's health, weapons, resources, troops, *etc* [26]. This cheat is possible as developers often incorrectly assume that the client software can be trusted not to reveal secrets. Secret information is revealed by either modifying the client or running another program that extracts it from memory. One of the most prolific examples is the map hack in Real Time Strategy (RTS) games such as Warcraft III [2]. Information exposure also occurs at the infrastructure level using different methods of attack.

The most effective solution to prevent this cheat in C/S architectures is using On Demand Loading (ODL) [19]. Using this technique a trusted third party (the server) stores all secret information and only transmits it to the client when they are entitled to it. Therefore, the client does not have any secret information that may be exposed. P2P architectures can only use ODL if there is a trusted third party – such as a server – to store the secret information. RACS uses the referee to store secret information [33].

4.2.2 Bots/reflex enhancers

This form of cheat requires modifying the game client or running an external program to generate user input. Bots such as WoW Glider [20] use computer AI to completely control the player's avatar to automate repetitive tasks, progressing the player's avatar through the game. Reflex enhancers merely augment user input in reflex games to achieve better results. FPS such as Halflife often suffers from reflex enhancers that automatically aim at opponents [9]. Both C/S and P2P architectures are vulnerable to this form of cheating.

To prevent bots/reflex enhancers many new games require running a cheat detection application such as PunkBuster (PB) [27] or Valve Anti-Cheat 2 (VAC2) [32] that scans the player's host memory searching for cheating applications. These programs match checksums of running applications against a database of known cheats to detect cheating. PB and VAC2 can be used in both C/S and P2P architectures. Another alternative is to use statistical analysis to detect cheating [37]; however, by introducing randomness into a bots aim a cheater may go undetected [26].

4.2.3 Invalid commands

Usually implemented by modifying the game client, the invalid command cheat results in the cheater sending commands that are not possible with an unmodified game client. Examples include giving the cheater's avatar great strength or speed. This may also be implemented by modifying the game executable or data files. Many games suffer this form of cheating, including console games such as Gears of War [11]. The invalid command cheat is

easy to prevent in C/S architectures or RACS as the server or referee simulates and validates all commands, and can be trusted to produce the correct result. However, preventing invalid commands in P2P is difficult as there is no trusted entity to verify the commands. The solution is to build some form of trust amongst the peers and then use the trusted peers to validate the simulation. For example, the validation peer could be selected randomly, and without any vested interest in the outcome of the simulation. Using a group of peers achieves better security, because if only one peer is used a griefer may be able to disrupt the game. Group selection is mentioned in [8]; however, the authors leave it for future work.

Mönch *et al*. [21] propose using tamper resistant techniques to prevent modifications to the game client; hence, preventing invalid commands. Although this does not prevent cheating; it can make it significantly more difficult. Furthermore, if successful this approach prevents some forms of information exposure and proxy/reflex enhancers (see Section 4.4.2). The cost is significant additional processing on the clients, and developing tamper proof software is a non-trivial task for the developer. We are not aware of any games using this technique; therefore, it is difficult to evaluate.

4.3 Protocol level cheats

Protocol level cheats involve interfering with the packets sent and received by the game. Packets may be inserted, destroyed, duplicated, or modified by an attacker. Many of these cheats are dependent on the architecture used by the game (C/S or P2P). In the following, we describe seven protocol level cheats: suppressed update, fixed delay, timestamp, collusion, blind opponent, and undo.

4.3.1 Suppressed update

As the Internet is subject to packet loss most networked games use dead-reckoning [1]. In the event of a lost/delayed update the server will extrapolate (dead-reckon) the players movement from their current position, creating a smooth movement for all other players. Dead-reckoning usually allows clients to drop up to *n* consecutive packets (which are dead-reckoned) before they are disconnected. In the suppressed update cheat, a cheater purposely does not send up to *n*-1 consecutive updates, while still accepting opponent updates. Before the n^{th} update the cheater calculates the optimal move using the updates from their opponents and transmits it to the server. Thus, the cheater knows their opponents actions before committing to their own, allowing them to choose the optimal action. Although we are not aware of any real world occurrences of this cheat, it is potentially possible for most FPS, and any game – either C/S or P2P – that uses dead reckoning.

Architectures with a trusted entity (*e.g*., server), such as C/S or RACS, prevent this cheat by making the server's dead-reckoned state authoritative. Players are forced to follow the dead-reckoned path in the event of lost/delayed updates. This gives a smooth and cheat free game for all other players; however, it will disadvantage players with slow or lossy Internet connections. As a slow or lossy Internet connection is already a major disadvantage [10] we believe this will not have a significant impact.

Cronin *et al.* [10] propose the Sliding Pipeline (SP) protocol to prevent this cheat in P2P architectures. In SP players constantly monitor the delay to their opponents and compare it with the timestamps of updates. Late updates indicate that a player is either suffering delay, or is cheating. The authors claim that this protocol will detect all cheaters, but acknowledge that players with poor connectivity may be falsely detected as cheaters (false positive).

4.3.2 Fixed delay

This form of cheat was discovered in Madden NFL Football by Nichols and Claypool [24], but was not proposed as a method of cheating until [14]. Fixed delay cheating involves introducing a fixed amount of delay to all outgoing packets. This results in the local player receiving updates quickly, while delaying information to opponents. For fast paced games this additional delay can have a dramatic impact on the outcome. This cheat is usually used in P2P games when one peer is elevated to act as the server; thus, they can add delay to all other peers. To prevent this cheat P2P games should use distributed event ordering and consistency protocols to avoid elevating one peer above the rest (See Section 5). Note, the fixed delay cheat only delays updates, in contrast to dropping them in the suppressed update cheat.

4.3.3 Inconsistency

Specific to P2P architectures, a cheater induces inconsistency amongst players by sending different game updates to different opponents. An honest player attacked by this cheat may have his game state corrupted, and hence be removed from the game, by a cheater sending a different update to him than was sent to all other players. This cheat may also be used by a cheater or group of cheaters to gain an unfair advantage, and later merged with the other player's game state to make it undetectable [14].

To prevent this cheat updates sent between players must be verified by either a trusted authority, or a group of peers. In RACS [33] the referee receives hashes of every update sent between peers which it uses to detect the inconsistency cheat. This is possible as the referee is a trusted entity. In P2P protocols without a trusted 3^{rd} party the group must form a consensus about which updates are valid. The consensus is achieved by voting on the hashes of updates of all players; however, group selection is critical as several colluding cheats could potentially bias the group vote [8]. SEA discusses using group verification; however, it does not provide a group selection mechanism; therefore, the solution is incomplete [8].

4.3.4 Timestamp

This cheat is enabled as many games allow un-trusted clients to timestamp their updates for event ordering. This allows cheaters to timestamp their updates in the past, after receiving updates from their opponents; hence, they can perform actions with additional information honest players do not have. C/S and RACS avoid this problem by using the arrival order of updates to the server for time stamping [14,33]. Alternatively the proposal [7] uses active RTT measurements between the server and peers to detect cheating in C/S architectures. See Section 5 for known solutions in P2P protocols.

4.3.5 Collusion

Collusion involves two or more cheaters working together (rather than in competition) to gain an unfair advantage. Colluders often communicate via an external channel – over the phone, instant messaging, VoIP, *etc*. Collusion is extremely difficult or impossible to detect/prevent and has far reaching ramifications. There are many examples of collusion in networked computer games; however, one common example is of players participating in an all-against-all style match, where two cheaters will team up

(collude) against the other players. This occurs in both C/S and P2P, and is effectively undetectable, although there are novel approaches to detect this cheat in [35].

4.3.6 Spoofing

Spoofing is a traditional network security threat where a cheater sends a message masquerading as a different player [8]. For example, a cheater may send an update causing an honest player to drop all of their items. To prevent this cheat in both C/S and P2P, updates should be either digitally signed or encrypted. With either technique the receiver can validate the senders identity. We are not aware of any real-world games where it has occurred even though most games are vulnerable to spoofing.

4.3.7 Replay

If a cheater receives digitally signed/encrypted copies of an opponent's updates he may be able to disadvantage an opponent by resending them (replay) at a later time [8]. As the updates are correctly signed or encrypted they will be assumed valid by the receiver. To prevent this in C/S and P2P updates should include a nonce (unique number), such as a round number or sequence number. When an update is received the receiver should check to ensure the nonce is fresh (has not been used before). While many games are vulnerable to replay attacks, we are not aware of any examples where this cheat has been used.

4.3.8 Blind opponent

A cheater may purposely drop updates to opponents, blinding them about the cheaters actions, while still accepting updates from opponents [33]. This cheat is only possible in some P2P protocols [25]. A tit-for-tat scheme where players stop sending updates to cheaters - effectively blinding the cheater as well - is an insufficient solution for this cheat as there are instances where dropping updates would still give the cheater an advantage, such as if they need to make a retreat. We are not aware of any real world instances where this cheat has been used. P2P solutions are discussed in Section 5.

4.3.9 Undo

Some P2P protocols [1,8,14] use a commit/reveal scheme to prevent the suppressed update, fixed delay, timestamp, and blind opponent cheats; however, if the reveal step is not enforced (as in [8,14]) it is possible for a cheater to reveal their opponent's move and asses it, before deciding if they will reveal their move. If a cheater does not reveal their move they effectively undo the move. P2P protocols that require all updates to be revealed (*e.g.,* Lockstep and AS) or do not use the commit/reveal process (*e.g.,* RACS) are immune. This cheat was first discussed in [33].

4.4 Infrastructure level cheats

Infrastructure cheats involve modifying or interfering with the software (*e.g.*, display drivers) or hardware (*e.g.*, the network infrastructure) that the game is using. In the following we describe two examples of infrastructure level cheats: information exposure, proxy/reflex enhancers.

4.4.1 Information Exposure

Information exposure (infrastructure level) is applicable to both C/S and P2P and is enabled by modifying either the client's network or display drivers. If data is broadcast across the network, (*e.g.,* when using a non-switching hub) a cheater can use a different host to sniff network traffic intended for his host, which it then displays to the cheater. ShowEQ [28] is one example that captures and interprets Everquest traffic. Alternatively, by modifying the display drivers to render the world differently, such as with transparent walls, a cheater gains access to secret information such as the locations of opponents.

As sniffing network traffic is entirely passive and does not take place on the cheater's computer it is impossible to detect packet sniffing. However, PunkBuster [27] and VAC2 [32] can be used to detect modified drivers by scanning them for modifications; this solution can be used in both C/S and P2P architectures. As previously described, On Demand Loading (ODL) is the most effective countermeasure against information exposure in both application and infrastructure levels.

4.4.2 Proxy/Reflex Enhancers

Reflex enhancers are implemented at the infrastructure level by deploying a proxy between the client and the server to modify the client's packets. As commands pass through the proxy it will insert or modify commands to improve the cheaters actions. Quake was one of the first games to suffer from aiming proxies, where the proxy inserts a movement command immediately preceding all shoot commands to aim at the nearest opponent [26].

This cheat effects both C/S and P2P, and there is no complete solution. One proposal is to compare checksums of the client and server states; however, as many games use UDP this is often not possible. Alternatively encryption can be used; however, the client cannot be trusted to keep the key secret; therefore, the encryption will be broken.

5. P2P CHEAT PREVENTION PROTOCOLS

Several P2P protocols have been proposed to prevent cheating. In this section we discuss the strengths and weaknesses of each. With the exception of RACS, these protocols only attempt to address protocol level cheating, ignoring the information exposure and invalid command cheats that are prevalent in network games and preventable using C/S [33].

5.1 Lockstep

Lockstep [1] divides game time into rounds and requires that every player in the game submit their move for that round before the next round is allowed to begin. To prevent cheating, all players commit to a move, and once all players have committed, each player reveals their move. A player commits to a move by transmitting either the hash of a move or an encrypted copy of a move, and it is revealed by sending either the move or encryption key respectively. Lockstep is provably secure against all protocol level cheats except the inconsistency cheat, as it does not use digital signatures to authenticate updates; hence, the cheat cannot be verified. Lockstep is also unacceptably slow for many fast paced games, with a worst case delay of $3d$; where d is the delay between the two slowest players

5.2 Asynchronous Synchronization

Asynchronous Synchronization (AS) [1] relaxes the constraints of Lockstep, only requiring players to work in Lockstep with the other players within their AoI. This greatly increases the speed the game can progress; however, AS is still slow, with the round length being at least twice the delay between the two slowest players within each other's AoI, and upper bounded by three

times the delay. Furthermore, it is also possible for a griefer to increase the delay intentionally to reduce the game play experience of other players. As with Lockstep, AS prevents all protocol level cheats except the inconsistency cheat.

5.3 Sliding Pipeline

The Sliding Pipeline (SP) [10] protocol is another extension of Lockstep, allowing updates to be pipelined and dead-reckoning to be used; thus, improving the smoothness of the game. SP works by constantly monitoring the delay between players to determine the maximum allowable delay for an update without allowing timestamp cheating. Unfortunately, SP cannot differentiate between players suffering delay and cheaters (false positives). Further, the worst case scenario remains at 3*d*, where *d* is the delay between the two slowest players. SP solves the timestamp, suppressed update cheats, and blind opponent cheats.

5.4 NEO/SEA

The New Event Ordering (NEO) protocol [14] explicitly bounds the round length to 2*d*. Players must be able to send updates to more than half of the group within *d* time to have the update accepted as valid. Any late updates are discarded. Players then transmit their key in the second half of the round. To increase responsiveness rounds may be pipelined. NEO is effective in preventing malicious players eroding the game-play experience beyond a pre-defined limit (2*d*), and also provides functionality to re-negotiate the round length, increasing the responsiveness of the game. The Secure Event Agreement protocol (SEA) [8] is an update to NEO with modified cryptographic techniques. The authors of SEA demonstrate that NEO is still vulnerable to several forms of cheating (replay attack, spoofing, and inconsistency), and improve the security and performance by changing the cryptography. Both NEO and SEA are bounded by 2*d*. As NEO and SEA do not force players to reveal moves that have been committed, both protocols are vulnerable to the undo cheat. SEA is secure against all other protocol level cheats.

5.5 Referee Anti-Cheat Scheme

Webb, *et al.* [33] propose the Referee Anti-Cheat Scheme (RACS), a C/S and P2P hybrid that increases the scalability of C/S without reducing its security. RACS uses a trusted central server (the referee) to receive, simulate, and validate all client updates to prevent cheating. To increase scalability RACS allows peers to exchange updates directly, reducing the referees outgoing bandwidth and processing requirements. Furthermore, as updates are not routed through the referee the delay between peers is minimized; thus, improving responsiveness. RACS uses two communication models: Peer-Referee-Peer (PRP) and Peer-Peer (PP). In PRP mode all updates are routed through the referee – as in C/S. In PP mode updates are sent directly between peers and a copy to the referee. RACS penalizes cheaters and slow players by forcing them to use PRP mode, increasing their delay. Note that a delayed packet may be coming from a slow player (due to network delay) or from a cheater (fixed delay or suppressed update cheat), and is arguably difficult to differentiate [1]. Even though RACS cannot differentiate between a cheater and a slow player, this penalty in essence is equivalent to the cheating-evident systems of [8,10,14]. RACS directly prevents the following cheats: information exposure, invalid commands, suppressed update, timestamp, fixed delay, inconsistency, replay, spoofing, and blind opponent. To be effective the game developer must also release updates to prevent bug cheats, and combine RACS with PunkBuster or VAC2 to prevent bots/reflex enhancers. However, preventing collusion and proxy/reflex enhancers is impossible in C/S, RACS, or P2P. While RACS increases the scalability of C/S by reducing the outgoing bandwidth and lowering the number of AoI calculations; the referee may still be a bottleneck as it must receive all player updates and simulate the entire world. Furthermore, the referee is a single point of failure for the system.

6. CONCLUSION

We have shown that cheating is prevalent, wide spread, and evolving in online games. Cheating is a major obstacle that must be prevented for an online game to be successful. We have extended the classification of cheating in [29] to include new forms of cheating, and to include real-world examples. Finally we have discussed various P2P protocols that prevent cheating.

While the referee concept in RACS has proved successful in reducing the server's outgoing bandwidth, it does not address the incoming bandwidth or processing bottlenecks. Reference [34] uses mirrored referees to address the incoming bandwidth issue. Our next goal is to perform load balancing amongst referees to remove the processing bottleneck, greatly increasing scalability. Our final goal is to distribute referees to peer machines, removing the incoming bandwidth bottleneck. The resulting P2P architecture will have high scalability while maintaining the security of C/S. However, this raises issues of referee trust, selection, load balancing, and synchronization. These issues require further investigation.

7. REFERENCES

[1] Baughman, N. E., Liberatore, M., & Levine, B. N. *Cheat-Proof Playout for Centralized and Peer-to-Peer Gaming*. IEEE/ACM Trans. Networking 22, **1** (2007), pp. 1-17.

[2] Blizzard. *Map Hack*. Web page. http://www.blizzard.com/support/?id=nNews054p, Aug. 2002.

[3] Blizzard. *World of Warcraft®: The Burning Crusade™ continues record-breaking sales pace*. Press release. http://www.blizzard.com/press/070307.shtml, Mar. 2007.

[4] Brandt, D. *Networking and Scalability in EVE Online*. Slide Show. http://www.research.ibm.com/netgames2005/papers/brandt.pdf, Oct. 2005.

[5] Caldwell, P. *Blizzard bans 59,000 WOW accounts*. Article. GameSpot AU. http://au.gamespot.com/pc/rpg/worldofwarcraft/news.html?sid=6154708, Jul. 2006.

[6] Chambers, C., Feng, W., Sahu, S., & Saha, D. *Measurement-based characterization of a collection of on-line games*. In IMC'05, Berkeley, CA, USA. pp. 1-14.

[7] Chen, B., Maheswaran, M., A cheat controlled protocol for centralized online multiplayer games, Netgames 2004, pp.

[8] Corman, A. B., Douglas, S., Schachte, P., & Teague, V. *A Secure Event Agreement (SEA) protocol for peer-to-peer games*. in Proc. *ARES'06,* pp. 34-41.

[9] Counter Hack. *HalfLife*. Web page. http://wiki.counter-hack.net/halflife, Mar. 2007.

[10] Cronin, E., Filstrup, B., & Jamin, S. *Cheat-Proofing Dead Reckoned Multiplayer Games*. in Proc. *Int. Conf Appl. Development of Computer Game*. 2003.

[11] Davis, S. *Next-Gen Hacking / Last-Gen Weaknesses - Part 1 - Gears of War for the Xbox 360.* Web page. http://playnoevil.com/serendipity/index.php?/archives/1123-Next-Gen-Hacking-Last-Gen-Weaknesses-Part-1-Gears-of-War-for-the-Xbox-360.html. Feb, 2007.

[12] DeLap, M., *et al.*, *Is runtime verification applicable to cheat detection?* in Proc. ACM *NetGames '04,* pp. 134-138.

[13] Franke, J. *Kingpin speed hack*. Web page. http://jjaf.de/kingpin/speed-hack/hack.html, Sep, 2000.

[14] GautheierDickey, C., Zappala, D., Lo, V., & Marr, J. *Low-Latency and Cheat-proof Event Ordering for Distributed Games*. in Proc. *NOSSDAV '04*, pp. 134-139.

[15] Kabus, P., Terpstra, W. W., Cilia, M., & Buchmann, A. P. *Addressing cheating in distributed MMOGs*. in Proc. *NetGames '05,* pp. 1-6.

[16] Knutsson, B., Lu, H., Xu, W., & Hopkins, B. *Peer-to-Peer Support for Massively Multiplayer Games*. in *INFOCOM '04, Hong Kong*, **1**: pp. 7-11.

[17] Kushner, D. *Engineering EverQuest: online gaming demands heavyweight data centers*. IEEE Spectrum 42, **7** (2005), pp. 34-39.

[18] Lee, J. *Wage Slaves.* Article. 1UP.com. http://www.1up.com/do/feature?cId=3141815, May, 2005.

[19] Li, K., Ding, S., McCreary, D., & Webb, S. *Analysis of state exposure control to prevent cheating in online games*. in Proc. ACM *NOSSDAV '04,* pp. 140-145.

[20] MDY Industries. *Glider*. Web page. http://www.wowglider.com/. 2007.

[21] Mönch, C., Grimen, G, and Midtstraum R, *Protecting online games against cheating*, Netgames'06, pp. 1-11.

[22] Mulligan, J., & Patrovsky, B. *Developing Online Games: An Insider's Guide*. 2003: New Riders Publishing.

[23] Neumann, C., Prigent, N., Varvello, M., & Suh, K. *Challenges in peer-to-peer gaming*. ACM SIGCOMM Computer Communication Review. 37, **1** (2007), pp. 79-82.

[24] Nichols, J., & Claypool, M. *The effects of latency on online Madden NFL football*. In Proc. ACM *NOSSDAV'04*. pp. 146-151.

[25] Pellegrino, J. D. & Dovrolis, C. *Bandwidth requirement and state consistency in three multiplayer game architectures*. in Proc. *NetGames '03,* pp. 52-59.

[26] Pritchard, M., *How to Hurt the Hackers*, in Game Developer Magazine, Jun. 2000. pp. 28-30.

[27] PunkBuster. *PunkBuster for players, Quake 4 edition*. http://www.punkbuster.com/publications/q4-pl/index.htm, Oct. 2005.

[28] ShowEQ. http://www.showeq.net/.

[29] Stutzbach, D., Stutzbach, R., & Sen, S. *Characterizing Unstructured Overlay Topologies in Modern P2P File-Sharing Systems*. In IMC'05, Berkeley, CA, USA. pp. 49-62.

[30] Terdiman, D. *Virtual good, real scams*. Article. ZDNet. http://news.zdnet.com/2100-1040_22-5859069.html, Sep. 2005.

[31] Terdiman, D. *'World of Warcraft' battles server problems*. Article, CNET News.com. http://news.com.com/World+of+Warcraft+battles+server+problems/2100-1043_3-6063990.html, Apr. 2006.

[32] Valve. *Valve Anti-Cheat System (VAC)*. Web page. http://support.steampowered.com/cgi-bin/steampowered.cfg/php/enduser/std_adp.php?p_faqid=370, Apr. 2007.

[33] Webb, S., Soh, S., & Lau, W. *RACS: a Referee Anti-Cheat Scheme for P2P gaming*. To appear in NOSSDAV '07.

[34] Webb, S., Soh, S., & Lau, W. *Enhanced mirrored servers for network games*. Submitted to ACM NetGames 2007.

[35] Yan, J. *Security Design in Online Games*. in Proc. IEEE *ACSAC '03,* pp. 286-295.

[36] Yan, J. & Randell, B. *A systematic classification of cheating in online games*. In Proc. ACM *NetGames '05,* pp. 1-9.

[37] Yeung, S., Lui, J., Liu, J., & Yan, J. *Detecting cheaters for multiplayer games: theory, design and implementation.* In Proc IEEE CCNC'06. Volume 2, pp. 1178-1182.