CMSC 425: Lecture 23
Detecting and Preventing Cheating in Multiplayer Games

Reading: This lecture is based on the following articles: M. Pritchard, "How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It," Gamasutra 2000; J. Yan and B. Randell, "A Systematic Classification of Cheating in Online Games," NetGames 2005, 1–9; S. D. Webb and S. Soh, "Cheating in Networked Computer Games: A Review," DIMEA 2007, 105–112

Cheating in Multiplayer Games: "Cheating" is defined to be acting dishonestly or unfairly in order to gain an advantage. In online games, players often strive to obtain an unfair advantage over others, for various reasons. One of the first analyses of cheating in online games appeared around 2000 in a Gamasutra article by Matthew Pritchard. He makes the following observations:

- If you build it, they will come to hack and cheat
- Hacking attempts increase as a game becomes more successful
- Cheaters actively try to control knowledge of their cheats
- Your game, along with everything on the cheater's computer, is not secure—not memory, not files, not devices and networks
- Obscurity $\neq$ security
- Any communication over an open line is subject to interception, analysis and modification
- There is no such thing as a harmless cheat
- Trust in the server is everything in client-server games
- Honest players would like the game to tip them off to cheaters

Pritchard identifies a number of common cheating attacks and discusses how to counter them. His list includes the following:

Information Exposure: Clients obtain/modify information that should be hidden.

Reflex Augmentation: Improve physical performance, such as the firing rate or aiming

Authoritative Clients: Although the server should have full authority, some online games grant clients authority over game execution for the sake of efficiency. Cheaters then modify the client software.

Compromised servers: A hacked server that biases game-play towards the group that knows of the hacks.

Bugs and Design Loopholes: Bugs and design flaws in the game are exploited.

Infrastructure Weaknesses: Differences or problems with the operating system or network environment are exploited.

We will discuss some of these in greater detail below.

Reflex Augmentation: Reflex augmentation systems involve the use of software that, through various methods, circumvents the user-based aiming/firing systems to a software-based system.

One example is an *aimbot*. An aimbot is implemented by modifying the game client program or running an external program in order to generate simulated user input. Network packets are intercepted and interpreted to determine the location of enemies and obstacles. Then computer AI is used to completely control the player's avatar and automate repetitive tasks, progressing the player's avatar through the game. Another example is a *reflex enhancer*, which augment a user's input in reflex games to achieve better results. For example, in a shooter game, the software can automatically aim at opponents.

Reflex augmentation typically involves modifying the underlying game executable or modifying one of the system's library functions that the game invokes. Techniques borrowed from the area of virus detection can be employed to be sure that the user has not tampered with the game's binary executable. Some approaches are static, using fingerprinting to scan the player's host memory in search of bit patterns of known cheating applications. A more dynamic approach is to periodically download the original game executable and compare its behavior to the user's game's behavior. If the executable has not been tampered with, then the two should behave identically. If not, the user must have tampered with the code somehow.

Information Exposure: This method of cheating involves the cheater gaining access to information that they are not entitled to, such as their opponent's health, weapons, resources, troops. 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.

Key Variables As a concrete example, let's consider cheating/hacking software that enables a hacker to access and modify the memory where the program stores its data. Your game has a integer variable, say num_lives, that controls the number of remaining lives the player has. A hacker wants to control this value but does not know where in memory this variable resides.

The hacker can employ the following trick to find it. Suppose that the player starts with 5 lives, so num_lives = 5. Search memory for all occurrences of memory locations containing the value 5. There are many of them. Next, kill yourself, thus reducing the number of lives to 4 and search among the previous hits to find those that now store the value 4. (This may sound tedious, but it is easy to design a program to help with this.) Repeat this until you can identify a unique (or generally small number of) match the above searches. Now, the hacker just modifies these memory locations to any desired value, say 9999, and now the hacker has infinite life.

How do we fix/detect this? One fix is to wrap important variables like num_lives within a class, say, EncryptedInt, where the actual value is stored in encrypted form. The getters/setters of the class decrypt/encrypt the value during each access. It is now much harder for the hacker to locate the memory location where this value is stored, and even if it could be found, it would be impossible (without hacking the encryption algorithm) to know what value to set it to.

Graphics Hacks: Suppose that you have a shooter game, where enemies may hide behind walls, bushes, or may rely on atmospheric effects like smoke or fog. The cheater then modifies the parameters that control these obscuring elements, say by making walls transparent, removing the foliage on bushes, and changing the fog parameters in the graphics system so it effectively disappears. The cheating application alone now has an un-obscured view of the battlefield.

Physics Hack: Unity uses physics components such as the RigidBody to keep a player from moving through obstructions like walls. One form of an attack involves inserting code that disables a player's or walls' RigidBody components, thus enabling this player to move through walls. (You need to watch out for gravity, less you fall through the floor as well.) While such an attack may be hard to prevent, it may be detectable by inserting code the periodically verifies that the program's key objects satisfy their key properties.

User-Preferences: In order to save data from one program execution to the next, such as user-preferences and earned assets, some operating systems (notably Microsoft Windows) save persistent data in a single database of saved data. In the case of Windows, this is called the *registry*. A hacker can look for an obviously named registry entry (e.g., "NUM_LIVES") and then employ a registry editor program (e.g., Windows regedit) to modify these values. Using obscure names and encrypting the values are prevent this.

Speed Hack: A *speed hack* modifies the game's perception of time to either slow-down or speed-up the game. This can be done to fast-forward through lengthy boring sequences or slow down targets so they are easier to attack. A game program's concept of time is based on calls to the clock functions offered by the operating system. Through the use of general-purpose hacking tricks, it is possible to hijack these calls, thus modifying your game's concept of time (e.g., by making the value of Time.deltaTime larger or smaller by some factor).

These are sometimes called *infrastructure-level cheats*, since they involve accessing or modifying elements of the infrastructure in which the program runs. In a client-server setting, this can be dealt with is using a technique called on-demand-loading (ODL). 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.

As mentioned above, another approach for avoiding information exposure is to encrypt all secret information. This makes it difficult to determine where the information is and how to interpret its meaning.

Protocol-level cheats: Because most multiplayer games involve communication through a network, many cheats are based on interfering with the manner in which network packets are processed. Packets may be inserted, destroyed, duplicated, or modified by an attacker. Many of these cheats are dependent on the architecture used by the game (client-server or peer-to-peer). Below we describe some protocol-level cheats.

Suppressed update: As we mentioned last time, the Internet is subject latency and packet loss. For this reason, most networked games use some form of dead-reckoning. In the event of a lost/delayed updates, the server will extrapolate (dead-reckon) the players movement from

their most recent position and velocity, creating a smooth movement for all other players. Dead-reckoning usually allows clients to drop some fixed number of consecutive packets before they are disconnected. In the suppressed update cheat, a cheater purposely *suppresses* the transmission of some fixed number consecutive updates (but not so many to be disconnected), while still accepting opponent updates. The attacker (who is able to see the other player's movements during this time) 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.

Architectures with a trusted entity (e.g., the server), can prevent this cheat by making the server's dead-reckoned state authoritative (as opposed to allowing the client to do it). 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 may disadvantage players with slow Internet connections. In a less authoritative environment (e.g., peer-to-peer) it may be possible for other players to monitor the delay in their opponents and compare it with the timestamps of updates. Late updates indicate that a player is either suffering delay, or is cheating.

Fixed delay: 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 peer-to-peer games, when one peer is elevated to act as the server. Thus, they can add delay to all other peers.

One way to prevent this cheat in peer-to-peer games can use distributed event ordering and consistency protocols to avoid elevating one peer above the rest. Note, the fixed delay cheat only delays updates, in contrast to dropping them in the suppressed update cheat.

Another solution is to force all players to use a protocol that 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. (One such protocol is called *lockstep*.) 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 these and other protocol level cheats. Unfortunately, this approach is unacceptably slow for many fast-paced games, since it forces all players to wait on the slowest one.

Another example of a protocol to prevent packet suppression/delaying is called *sliding pipeline* (SP). SP works by constantly monitoring the delay between players to determine the maximum allowable delay for an update without allowing times-stamp cheating (see below). SP does not lock all players into a fixed time step, and so can be applied to faster-paced games. Unfortunately, SP cannot always differentiate between players suffering delay and cheaters (false positives).

More Protocol-Level Cheats: The above (suppressed update and fixed delay) are just two examples of protocol-level cheats. There are many others, which we will just summarize briefly here.

Inconsistency: 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. To prevent this cheat updates sent between players must be verified by either a trusted authority, or a group of peers.

Time-stamp: This cheat is enabled in games where an untrusted client is allowed to time-stamp their updates for event ordering. This allows cheaters to time-stamp their updates in the past, after receiving updates from their opponents. Hence, they can perform actions with additional information honest players do not have. To prevent this, rather than using timestamps, processing should be based on the arrival order of updates to the server.

Collusion: Collusion involves two or more cheaters working together (rather than in competition) to gain an unfair advantage. 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. Colluding players may communicate over an external channel (e.g., over the phone or instant messaging). This is very hard to detect and prevent.

Spoofing: Spoofing is where a cheater sends a message masquerading as a different player. For example, a cheater may send an update causing an honest player to drop all of their items. To prevent this cheat, updates should be either digitally signed or encrypted.

If a cheater receives digitally signed/encrypted copies of an opponent's updates he may still be able to disadvantage an opponent by resending them at a later time. Since the updates are correctly signed or encrypted, they will be assumed valid by the receiver. To prevent updates should include a unique number, such as a round number or sequence number, which the receiver can then check to ensure the message is genuine.

Detecting Cheating: Here are a few approaches to detect/prevent cheating.

- *Signature detection* - Detecting of certain patterns of bytes in memory, checked against a database
- *Heuristic analysis* - Statistical analysis of behavior
- *User reports* - Information provided by other users

Signature detection: This is the primary method in which cheating software is detected. Let's consider how this works for anti-virus software.

- A cheating attack (hack) is developed.
- The hacked software is analyzed, and a "signature" that identifies the binary of the hacked code is saved in a database.
- Before installing software, its binary is checked against the known signatures to detect hacked versions.

The difficulty in game programming is that, even though a cheater is caught, the hacked code resides on the cheater's machine, and hence may be difficult to obtain by the "authorities" who manage the game. This might be done, for example, by having a program that transmits a memory scan of the infected software, but this solution is

quite expensive and may not scale up to large numbers of users. But, this is assuming that the user actually designed his own hacked version of the game software. Most users are not this sophisticated, and instead download the sofware from third-party providers of hacked game software. The authorities can find these same sites, download the hacked software, and create signatures that will identify them.

Heuristic analysis: Lets say the you are a typical player. You have established a baseline of typical performance. Sometimes you have an excellent performance and do much better than your baseline. As your skill improves, your baseline trends upwards as well. Your performance can also become significantly worse (as when you loan your ID out to your younger cousin who is visiting for a few weeks). So, if your performance suddenly surges up and maintains a high level over a period of time, this would seem suspicious.

Heuristic analysis is an application of machine learning, in which an analysis of player statistics (scores, kill rates, speed and accuracy of mouse movement) to detect behavior that sufficiently far from the norm, that it can be infered to be the result of cheating. This is particularly useful against reflex augmentation software, provided that the software does "too good" a job of enhacing a player's performance. Of course, a careful cheater who understands this will keep his/her performance within believable limits, slowly increasing their baseline performance to avoid detection.

User reports: If a cheater's behavior is spoiling the experience for many users, some of these users will complain, and this can lead to detecting the cheater.