Server-Side Bot Detection in Massive Multiplayer Online Games

One of the greatest threats that massive multiplayer online games face today is a form of cheating called botting. The authors propose an automated approach that detects bots on the server side based on character activity and is completely transparent to end users.

STEFAN MITTERHOFER AND CHRISTIAN PLATZER
Vienna University of Technology

CHRISTOPHER KRUEGEL
University of California, Santa Barbara

ENGIN KIRDA
Eurecom, Sophia Antipolis, France

Massive multiplayer online games (MMOGs) have soared in popularity in the past few years, with a rapidly growing user base and game studios pouring tens of millions of dollars into developing their next big title. The market leader alone—Blizzard Entertainment's World of Warcraft (WoW)—surpassed 11.5 million subscribers in December 2008, raking in an estimated US$150 million in subscription fees per month. With such amounts of money at stake, it's not surprising that game companies want to keep their paying customers satisfied and threats to their revenue base at bay. One of these threats is *botting*, a form of cheating[1] in which players use a program that can play the game with a minimum of (or sometimes even zero) human interaction.

To the best of our knowledge, the only automated tool against bot programs is the Warden, an application that monitors WoW.[2] The Warden runs on a player's computer while he or she plays WoW and checks for suspicious programs such as debuggers or bots. It reports back to Blizzard, and any violations result in temporary or permanent account bans. However, the Warden has several shortcomings: it can only perform signature checks for known programs, which means it's always a step behind bot writers, and it runs on the client's computer, which is completely out of Blizzard's control. This ultimately means that its results can't be trusted. Additionally, players have already created some simple workarounds, such as starting the game in guest mode on an administrator account, which prevents the Warden from accessing the processes at higher privilege levels. Not surprisingly, privacy issues have also emerged.[3]

We propose a novel approach that relies solely on a server-side analysis of character (or avatar) behavior to expose bots and avoid many of the drawbacks found in client-side solutions. To this end, we exploit an intrinsic bot feature—namely, the fact that it's controlled by a script that automates a specific sequence of constantly repeated actions. We focus specifically on the game character's movement by extracting waypoints that describe the traveled path and finding repeated patterns in the route taken. (Here, a *route* is the course of movement that a character performs in the game world, and a *path* is a sequence of locations that the character visits; a route can follow the same path several times over.) We implemented and evaluated our approach in WoW.

How Bots Work

Players gravitate to bots because parts of a game can be inherently repetitive or boring. In particular, a player might need to kill large numbers of enemies to gain experience points and earn gold (a process called *farming* in the gaming community), which is often required to improve the character and progress further in the game. Running a farming bot means that the character reaps experience points and gold without the player investing any time in the game, as the bot can reap those rewards very efficiently 24 hours a day, without fatigue or boredom.

Interestingly, players don't use bots just to improve their own characters. There's a booming market for points, gold, and fully realized characters on the Inter-

Related Work in Game Bot Detection

Online gaming has only recently started to receive interest from the security research community. To date, there isn't much work on the detection of bots in online gaming.

One recent study[1] concentrates on simple bots as they exist in online games such as poker. The authors propose using CAPTCHAs that automated scripts can't solve to prevent bots from playing the game automatically. However, the proposed approach is strongly disruptive and has a heavy impact on the gaming experience for players—for example, many users might find it tedious and difficult to deal with CAPTCHAs and decide to quit playing the game. Our technique, in comparison, is intended for more complicated games, but it's completely transparent to the end user and has no influence at all on the gaming experience. Moreover, we focus on detecting bots, not preventing them from running on clients.

The most closely related work is by Kuan-Ta Chen and his colleagues, who used a traffic-analysis approach to identify bots for the game Ragnarök Online,[2] but their method's major drawback is that it's tailored for that specific game. Moreover, the authors attempted to distinguish traffic generated by the official game client from traffic generated by stand-alone bot programs through statistical analysis of packet transfer properties. Unfortunately, this approach no longer works for modern games: most of them use ping-independent command queuing on the client side (which changes network-level properties). In addition, most MMOG bots work by interacting with the official game client (for example, through code injection into the game process) and don't send any packets themselves. In contrast, our approach is aware of the game's semantics and leverages character behavior to identify bots. Thus, our technique is independent of traffic conditions and applicable to a wide range of MMOGs.

A good starting point for anyone interested in online game security is a recent book by Greg Hoglund and Gary McGraw that covers a wide section of game security topics, ranging from the legal issues over bug exploits and hacking game clients to writing bots.[3] Although the book provides a good introduction into several gaming security areas, it concentrates mostly on the attacker's viewpoint and doesn't provide concrete solutions on how to detect or prevent botting.

References

1. R.V. Yampolskiy and V. Govindaraju, "Embedded Noninteractive Continuous Bot Detection," *Computers in Entertainment*, vol. 5, no. 4, 2007, pp. 1–11.
2. K.-T. Chen et al., "Identifying MMORPG Bots: A Traffic Analysis Approach," *Proc. 2006 ACM SIGCHI Int'l Conf. Advances in Computer Entertainment Tech.*, ACM Press, 2006, article no. 4.
3. G. Hoglund and G. McGraw, *Exploiting Online Games: Cheating Massively Distributed Systems*, Addison-Wesley Professional, 2007.

net for those people who don't want to go through the hassle of obtaining them through hours of playtime. According to some estimates,[4] up to 400,000 people in China work as gold farmers for MMOGs, gathering gold and other in-game items and building up characters for the sole purpose of selling them online. Obviously, bots are even cheaper than human labor, making them the ideal tool for manipulating the game.

Botting's impact on a game can be substantial. In particular, it can lead to significant levels of inflation in the game economy because bots can, for example, create unusual amounts of gold by perpetually slaying monsters. This large amount of circulating gold in turn increases prices for items to the point where honest players can't afford them anymore, affecting their motivation to play at all and ultimately the game company's revenue. The Lineage game series suffered from this snowball effect.[5]

Considering botting's negative impact, it's not surprising that online game providers usually forbid its use or try to keep it at a minimum in their games. However, it's difficult to recognize bots because they typically obey game rules. When game providers take any action to detect bots in their virtual worlds, it's usually through human interaction—essentially, a mechanism that lets players report suspicious characters. When a player reports a suspicious character, a game moderator approaches the character in-game and tries to start a conversation, thus checking to see if a human is at the keyboard. However, this method is inefficient and doesn't scale well for worlds with tens or hundreds of thousands simultaneously online characters.

A more appropriate way to automatically detect bots is to focus on the server side, not the client side. Bots usually move around virtual worlds, killing nonplayer mobile entities (mobs). To achieve this, a player must specify a certain script that determines which actions the character chooses. Depending on the game situation, this can include the sequence of spells to cast in combat, under which circumstances the character heals itself, and whether it picks up loot from enemy corpses. The player additionally sets a path that the bot should follow and the enemies it should attack.

To execute a script, a bot must first collect information about its surroundings in the virtual world and then react to that data by sending commands to the game. Humans simply look at the computer screen to determine trees, mountains, and mobs, but bots must apply different methods to collect this information; currently, the most common way is through memory reading. Similar to a debugger, the bot program hooks into the running game process and reads the list of mobs and their locations directly from their representation inside the game's memory.

Although this works well for mobs, it isn't feasible to obtain the necessary terrain information this way because, based on map files, the game engine decides on the fly if the current step is possible or if an obstacle is large enough to block the character. Hence, the bot program would need to re-implement parts of the game engine and use a sophisticated route-finding algorithm to calculate a sensible movement route in advance. To avoid this problem, the player usually "teaches" the bot a certain path by running it him- or herself, with the bot recording waypoints that it will follow later to replicate the path.

Another way of getting information about the game's surroundings that works particularly well in WoW is by writing an add-on for the game's scripting engine (which offers a limited API to query game information). Although the add-on can gather the desired information with this method, it isn't possible to control the game through the API. Thus, the add-on outputs the information to the user interface by color-encoding certain pixels. A desktop automation scripting environment such as AutoIt (www.autoitscript.com/autoit3/) then reads and decodes these pixels, decides the necessary actions, and communicates them to the game client.

Our Analysis Mechanism

Contrary to human players, who roam freely in a virtual world, bots follow a prescribed or previously recorded list of waypoints. This suggests that although a human player's movements drawn on a two-dimensional map (a movement graph) will look more or less random, such a graph produced by a bot will show that it takes certain paths repeatedly. Bots usually follow paths more or less exactly, but they sometimes leave them to attack a monster or collect loot from a slain enemy. These small irregularities increase the difficulty of detecting them through movement analysis.

From looking at a movement graph, it's often straightforward for humans to recognize repeated movement patterns, but this solution doesn't scale to large populations of players. Our approach automatically detects if a bot is in control. To accomplish this, it first processes and transforms a character's movement data before it interprets the results of these steps to expose bots.

Collecting Data and Building a Route

When a player moves his or her character in a virtual world, the game client regularly sends packets with new coordinates to the server. Hence, the movement coordinates are readily available on the server side. We log the character's movement and use this game trace as the basis of our detection approach.

In the first processing step of our approach, we reconstruct the route that the character took in the game world simply by connecting the coordinate dots that arrive at the server. Then, we process the dots via the Douglas-Peucker line simplification algorithm.[6] The algorithm's output is a simplified curve that resembles the original curve within certain tolerance levels but consist of fewer vertices. In this way, we eliminate dot clusters that occur when multiple packets arrive in quick succession at the same location—for example, to update the rotation when the mouse turns a character. The simplification helps the waypoint extraction algorithm concentrate on areas where the dots accumulate because the character passed that point several times. The result is a route represented as an ordered sequence of dots as displayed in Figure 1a.

Extracting Waypoints

When a bot takes the same path several times, dots of different runs accumulate into clusters of high dot density. In our next step, we extract these clusters and create a waypoint around each one; this waypoint is an area of a fixed diameter centered on the cluster's base. We choose the diameter to be as small as possible but large enough that, if a bot passes a waypoint location on a path multiple times, we count the waypoint's area as passed in every run.

To do this, we use a custom clustering algorithm based on the common *k*-means algorithm,[7] in which cluster size is limited to waypoint size. As our proximity measure, we use the Euclidean distance because it's the most accurate measure for this purpose. By constraining cluster growth to waypoint size, we make sure that a waypoint always contains all the dots of the respective cluster and its area is passed every time the character moves through one of the dots. Finally, we remove overlapping waypoints by keeping the one with the larger number of dots.

As Figure 1b shows, we end up with a set of waypoints distributed over the entire length of the path that the character took—they're a little sparser in the less well-traveled sections of the path and denser along the "beaten track" because each run's dots accumulate there.

Abstracting the Route Representation and Finding Repetitions

Next, we use the set of waypoints to create an abstracted and simplified route representation. We do this by iterating over the initial sequence of dots and, for each dot, recording the corresponding waypoint that the bot passes along the way. When a dot has no waypoint, we simply move on. Because each waypoint is uniquely identified, a sequence of waypoints can describe a route, as Figure 1c shows. This description is called a *movement sequence*.

To tackle the challenge of finding repetitions in the movement sequence, we leverage previous work

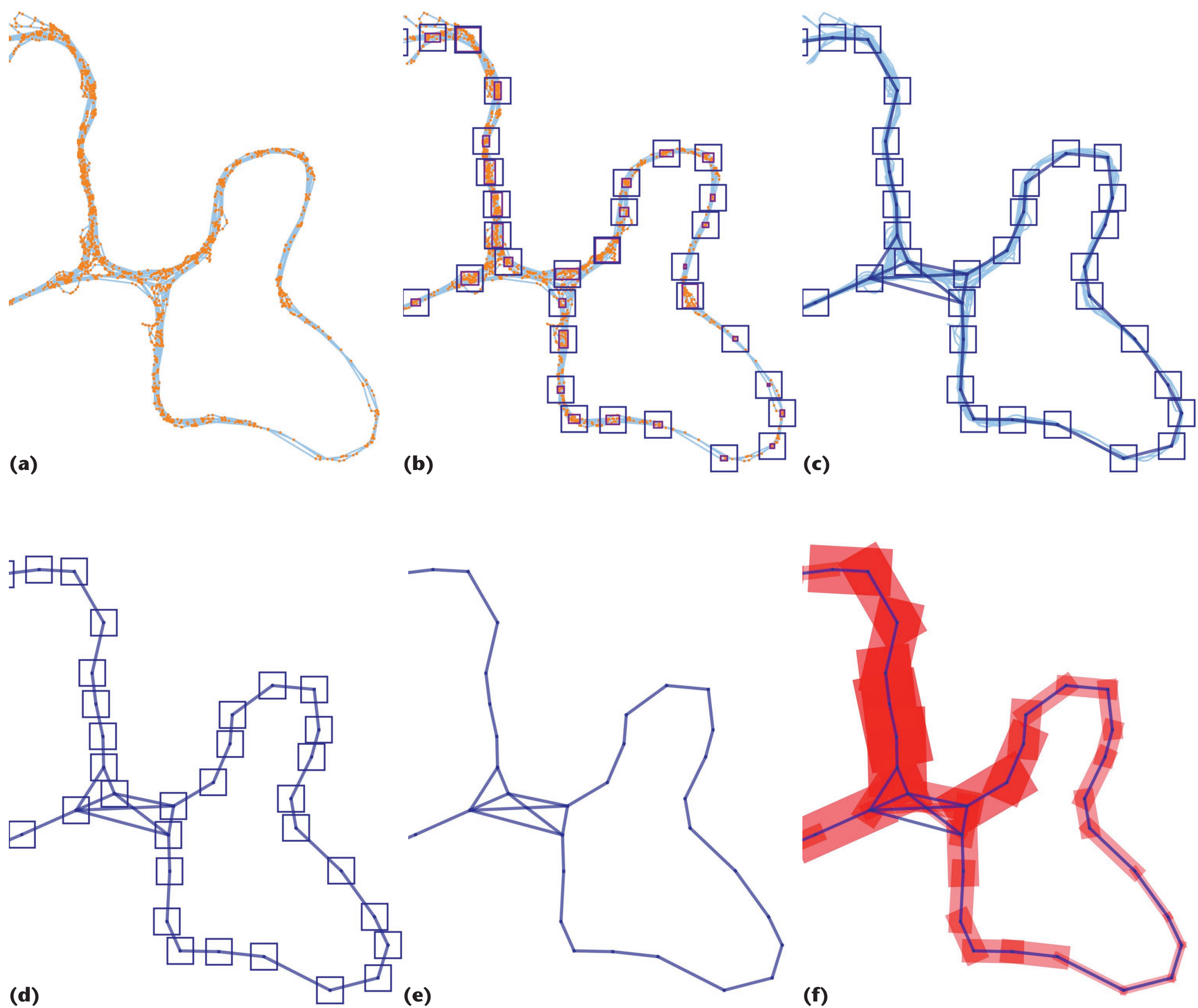

Figure 1. Movement data processing steps. (a) Route and dots, drawn after dot burst removal; (b) waypoints extracted from dot accumulations (the smaller purple rectangles mark the extracted clusters); (c) a simplified path (created by connecting waypoints with straight lines); (d) original route removed; (e) simplified path with waypoints removed; and (f) path segments show the number of times they were traveled.

in the area of bioinformatics. In this field, researchers have dedicated much effort to analyzing long sequences of DNA or proteins for interesting subsequences. We use an extended suffix array[8] to find repetitions in the movement (waypoint) sequence. A suffix array is a data structure for large texts that does some precalculations at creation time to facilitate and speed up subsequent search operations. At its heart lies an array of all suffixes of the text in alphabetical order. For the word "banana," for example, the sorted array consists of the suffixes {"a," "ana," "anana," "banana," "na," "nana"}.

We extend the suffix array by also calculating the longest common prefix (LCP) table, which contains the LCP that a suffix shares with the previous entry in the array. In other words, the LCP table shows how many characters two subsequent entries have in common before they differ, starting at the beginning of the suffixes. An LCP of five means that the suffixes are equal in their first five characters and differ in the sixth: because all suffixes are substrings starting at different positions in the original text, the substring of length five starting at these positions is a repetition. In the example suffix array for the word "banana," the LCP table for the entry "anana" would be three.

For our waypoint sequence, the LCP table lets us quickly look up repeated subsequences—in particular, a high LCP means that a long part of the route was repeated. In a perfect world, a bot would always pass the same waypoints in the same order, making it easy

to find the complete bot path from these repetitions, but a bot might choose not to run the full path in one go or won't follow the path exactly enough to always pass all the waypoints along the way. This is actually quite common—"perfect" runs are an exception, not the rule. The processing of LCP values helps us handle these inaccuracies and create a viable measure for subsequence repetitions.

Bot-Detection Metrics

The processing steps we've just described transform the movement data into intermediate results; we use them as measures for movement repetition to reliably differentiate between humans and bots. To actually raise bot alerts, we propose two metrics that interpret the intermediate results and decide whether the player is a bot: one takes into account how often a character travels through the locations it visits on its route, and the other looks at the amount and length of repeating subsequences in the route.

Decision based on average path segment passes. A path segment is a pair of adjacent waypoints in the movement sequence—a straight line from one waypoint until the character reaches the next one. The number of times a character passes a path segment shows how it has moved on that segment, independent of the direction. We calculate the number of average path segment passes by dividing the total number of path segment passes by the number of distinct path segments.

It's trivial to conclude that, when the character keeps moving on the same path, this number rises for the segments on that path. Because moving on the same path also means that no new waypoints are created (after all, a set of waypoints already describes the path), the number of distinct path segments remains constant, whereas the number of total and average path segment passes rises steadily. This is the typical case for bots, as seen in Figure 1f.

Human players, however, create new path segments continuously because random movement creates new segments between old waypoints or touches yet unexplored areas. Thus, the number of both distinct and total path segments keeps increasing, which leads to a low number of average path segment passes—a number that remains constant over the run of the game, usually well under two average passes per path segment.

By harnessing this difference, we can easily distinguish between bots and human players. Although the bot game traces show steadily rising numbers as the game commences, the number for the human game traces settles down at a constant low level. For our approach, we chose average path segment passes over average waypoint passes because we found the former to be a more robust indicator. Although both numbers describe the same principle and lead to similar graphs, path segment passes are significantly less likely to be repeated accidently. To pass a waypoint a second time, the character must visit the same location a second time. To pass a path segment for the second time, the character must be at the same location and move to the same waypoint next.

Decision based on repetitions in the waypoint sequence. Bots always take the same paths, and although they might not move through the exact same coordinates, their route still contains many and long repeating subsequences when compared to human routes. The LCP values from the waypoint sequence suffix array exactly capture this property. We calculate the average LCP value to ensure that the route must contain not only long but also many repetitions to display clearly suspicious numbers. To calculate the average LCP value, we sum up the LCP values of all entries in the suffix array and divide this by the number of entries. An average LCP of five means that for every waypoint in the waypoint sequence, the subsequence of the next five waypoints is repeated somewhere else. Humans typically never get close to the LCP values that bots reach because it's difficult to consistently keep passing the same waypoints in the same sequence. The average LCP for humans settles on a low level, whereas bot LCPs rise as they visit the same path over and over again. This provides a good discriminator between humans and bots.

Implementation and Evaluation

To evaluate our approach, we chose WoW, by far the most popular MMOG today. We started by setting up a WoW server and collecting game traces for two different bots and 10 different human players. For the server, we modified the ArcEmu open source WoW server emulator (http://arcemu.org) and logged all incoming and outgoing traffic and additional information about the game state in human-readable format to a text file.

Obtaining Traces

We prepared template game characters at level 12 for four character classes (warrior, paladin, mage, and warlock). To obtain human test data, we instigated a LAN party for an evening (four hours) and had seven people play the game on it, with three more joining over the Internet. The probands ranged from regular WoW players to beginners who had never played the game before. During the test gaming session, we told the players to concentrate on farming so that their gaming style was more likely to resemble that of bots.

To create bot game traces, we ran both the popular, subscription-based Glider bot (www.mmoglider.com)

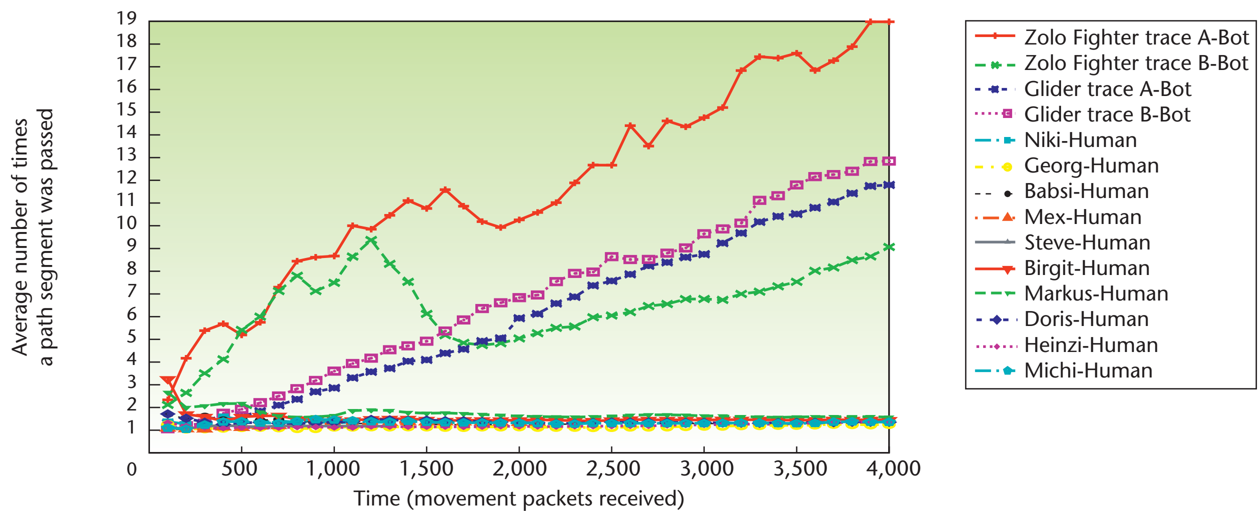

Figure 2. Average path segment passes. As bots only move on a preconfigured path over and over again, they keep passing the same segments, so this number rises. Humans rarely manage to move multiple times in the same pattern.

and the free ZoloFighter bot (www.zolohouse.com/wow/wowFighter), with the same template characters as the human players. We created two game traces per bot, using different bot paths each time. Blizzard recently sued the company that developed Glider, which clearly shows how seriously game providers are taking the botting threat (see http://forums.worldofwarcraft.com/thread.html?topicId=14910002728&sid=1).

The WOWalyzer

Next, we fed our traces into the WOWalyzer, a Java program we developed to visualize and analyze game traces based on the methods described earlier. As expected, the human movements looked random, with very few waypoints or path segments passed more than once; the bots, however, quickly showed repeating movement patterns. Depending on the length of the bot path and the number of enemies along the way, it took between 10 and 45 minutes until the beaten track was clearly visible from the movement graph.

Detection based on average path segment passes. Looking at the number of average path segment passes, the bot samples exhibited steadily increasing numbers, as Figure 2 shows, with the slope depending on the bot path's length. The noticeable dip of the ZoloFighter sample around time 1,800 owes its shape to the fact that a human player took over at roughly packet 1,200. The goal was to purposely disrupt the bot path and evaluate the influence on our detection mechanism. Later, at packet 1,800, we taught the bot a new path, which let the number of average line segment passes rise again. In sharp contrast, the human samples show steadily low numbers, settling down below two.

Detection based on repetitions in the waypoint sequence. To measure repeated subsequences in the waypoint sequences, we calculated the average LCP of the suffix array created from the waypoint sequence. As Figure 3 shows, the bot samples show significantly higher values than the human samples, because they contain a lot more as well as longer repeated subsequences. According to the graph, the human players didn't move in repeated patterns, which kept the average LCP on a stable low level below two, sometimes even below one.

Our Approach's Accuracy

The test results show that our technique can reliably distinguish between bots and humans. In general, we see that after a short time, the numbers for bots and humans start to diverge, as the bots begin repeating their movement pattern on their second run. The detection metrics trigger a bot alert whenever the number of average path segment passes or the average LCP reaches a certain threshold. In our implementation, we set the threshold for both metrics to five, which detects all bots within 12 to 60 minutes, while ensuring that no humans are falsely classified as bots. Once these values are reached, the trend continues, and the values never go back to normal as long as a bot is in control.

Our testing shows that it takes WOWalyzer less than two seconds to process four hours of gaming time on a single-core, 1.6-GHz Pentium-M. However, we propose using a sliding window of two hours to make the approach scale well; the time consumption of some processing steps rises at a quadratic rate. A shorter observation window also improves the approach's results for scenarios in which a human player plays for a while before switching over to a bot in

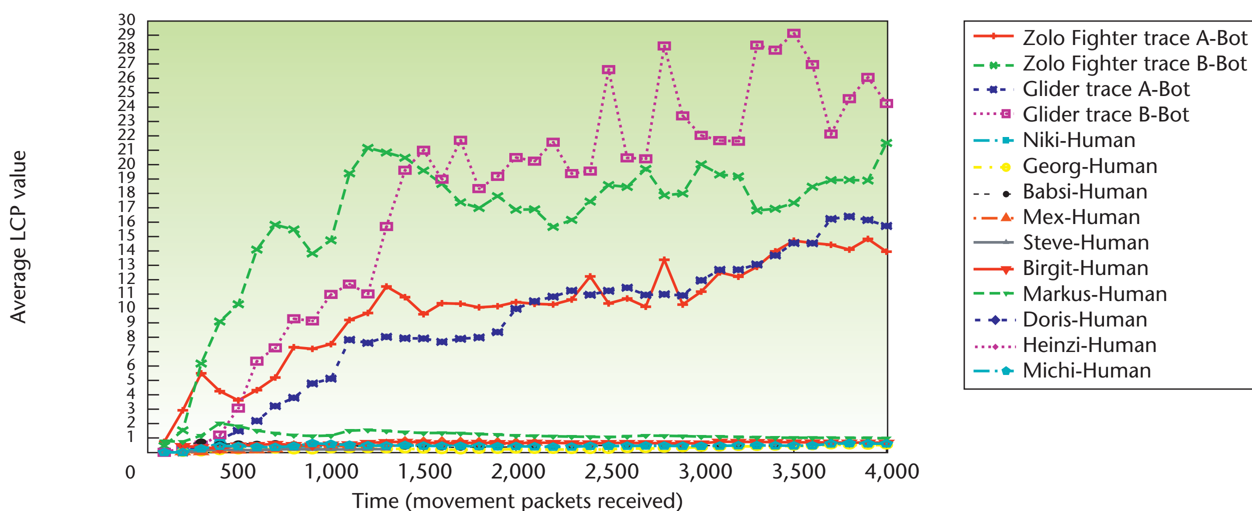

Figure 3. Average length of repeating waypoint subsequences (LCP). Bot movement patterns contain many and often long repeating subsequences, resulting in a high LCP. In contrast, humans hardly manage to repeat sequences of several waypoints in a row, thus their LCP stays at a stable low level.

an attempt to "hide" bot movements behind human activity. Once the human player's data slide out of the observation window, their diluting effect on the average numbers our approach calculates ceases.

Evading Detection

Our system is the ideal server-side bot-detection mechanism because it works completely transparently to the player, making it difficult for someone to discover what exactly triggered an account ban. This transparency also has the advantage of not impacting the gaming experience for honest players in any way. However, system discovery would ultimately lead to attempted countermeasures. Although our preliminary tests show that our system can distinguish between bots and humans, a bot could still avoid detection if it were tailored specifically to our approach's analysis mechanism.

One evasion vector for bots would be to use very long paths, which makes the number of repetitions rise more slowly. However, the player must create those long paths, which means that the amount of time that a person saves by using a bot decreases drastically. Simply using paths shared on the Internet increases the risk that those paths are also known to the game provider's bot-detection system.

Another way for bots to evade detection would be to move the character more randomly than it does already. However, the crucial point remains that all current bots for WoW use a navigation system that relies on waypoints, which inherently makes them susceptible to our and related approaches. Even though randomizing the movement along a path would impact the LCP values, the number of average line passes would hardly change. More general random movement would make the character susceptible to doing something stupid, thereby arousing suspicion from other human players.

Our tests with WoW prove that our approach effectively distinguishes between bots and humans and is computationally fast enough to monitor large numbers of players concurrently. Hence, we believe that our technique is an improvement over the sparse countermeasures currently in place in MMOGs.

To address the limitations we just mentioned and raise the bar for bot writers who aim to avoid detection, we intend to extend our system with statistical heuristics. Furthermore, we plan to test our approach on other MMOGs, such as Lineage and RuneScape. As long as bots don't replace their waypoint-oriented navigation with a different paradigm, we predict we can stay competitive in this arms race between bot programmers and game providers by continuously extending our method. ☐

References

1. J. Yan and B. Randell, "A Systematic Classification of Cheating in Online Games," *Proc. 4th ACM SIGCOMM Workshop on Network and System Support for Games*, ACM Press, 2005, pp. 1–9.
2. G. Hoglund, "4.5 Million Copies of EULA-Compliant Spyware," Oct. 2005; www.rootkit.com/blog.php?newsid=358.
3. C. McSherry, "A New Gaming Feature? Spyware," Electronic Frontier Foundation, Oct. 2005; www.eff.org/deeplinks/2005/10/new-gaming-feature-spyware.

4. R. Heeks, "Current Analysis and Future Research Agenda on 'Gold Farming': Real-World Production in Developing Countries for the Virtual Economies of Online Games," *Working Paper Series*, vol. 32, 2008; www.sed.manchester.ac.uk/idpm/research/publications/wp/di/di_wp32.htm.
5. Torak, "Bots: What *Is* NCSoft Doing?" IGN.com, 18 Oct. 2006; http://l2vault.ign.com/View.php?view=Articles.Detail&id=15.
6. D.H. Douglas and T.K. Peucker, "Algorithms for the Reduction of the Number of Points Required to Represent a Line or its Caricature," *The Canadian Cartographer*, vol. 10, no. 2, 1973, pp. 112–122.
7. H.-F. Eckey, R. Kosfeld, and M. Rengers, *Multivariate Statistics*, Gabler, 2002.
8. D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge Univ. Press, 1997.

Stefan Mitterhofer *is a master's student in software engineering and Internet computing at the Vienna University of Technology. His main research interests lie in the algorithms, inner workings, and security of distributed systems and Internet security. Mitterhofer is a student member of the IEEE. Contact him at sm@seclab.tuwien.ac.at.*

Christopher Kruegel *is an assistant professor in the computer science department at the University of California, Santa Barbara, and the holder of its Eugene Aas Chair in Computer Science. His research interests are computer and communication security, with an emphasis on malicious code analysis, Web security, and intrusion detection. Kruegel has a PhD in computer science from the Technical University Vienna. He also serves as an associate editor for the International Journal of Information Security. Contact him at chris@cs.ucsb.edu.*

Engin Kirda *is a faculty member of the networking and security department at the Institute Eurecom. His research interests are software and network security with a focus on Web vulnerability detection and prevention, binary analysis, and malware detection. Kirda has a PhD in computer science from the Technical University of Vienna. He is a member of the IEEE. Contact him at kirda@eurecom.fr.*

Christian Platzer *is an assistant professor in the computer-aided automation department at the Vienna University of Technology. His research interests are computer and communication security, with an emphasis on fraudulent behavior and spam-related topics. Platzer has a PhD in computer science from the Technical University of Vienna. Contact him at cplatzer@seclab.tuwien.ac.at.*