# SpotCheck: An Efficient Defense Against Information Exposure Cheats

Sam Moffatt
moffatts@cs.pdx.edu
Portland State University

Akshay Dua
akshay@cs.pdx.edu
Portland State University

Wu-chang Feng
wuchang@cs.pdx.edu
Portland State University

***Abstract*—A lot of hidden information is present in client programs of most existing online multi-player games. This hidden information is necessary for clients to render a player's view of the game. However, the same hidden information can be exploited by cheaters to gain an unfair advantage over other players. Eliminating hidden information from the game client comes at a significant cost to the server, since it must now send the data required to render a client's view on-demand. Consequently, the burden of tracking a player's view shifts from the client to the server, hindering scalability and degrading game performance. We propose SpotCheck, a more scalable approach for detecting information exposure cheats. The key idea is that servers still disseminate game state information on-demand, but clients retain the burden of tracking a player's view. After each move, clients must submit a descriptor pertaining to the player's view. The server then randomly chooses to validate the descriptor and sends back relevant game state information. Our experimental results show, that SpotCheck can reduce the server CPU overhead by as much as half when compared to the alternative, while still being an effective defense against information exposure cheats.**

## I. Introduction

Malicious access to hidden game client data allows cheaters to gain undue advantage in multi-player online games. Such cheats have been classified under the term *information exposure* [1], [2] and can be quite catastrophic. For example, in the real-time strategy game *Age of Empires*, a "map hack" allows the cheater to uncover map areas by modifying game client memory rather than actually exploring them, and to learn about secret resources and activities of opponents normally hidden to the player. In the words of Matthew Pritchard [2], "this cheating method was the next best thing to looking over the shoulders of his [the cheater's] opponents."

In client–server online games, information exposure cheats can be prevented by not introducing hidden state information in the game client. This can be done by having the server load game state information *on-demand* [11]. For example, the server could gradually expose only those portions of the game map a player can 'see'. Unfortunately, this method introduces game rendering delays because clients now need to retrieve state information from the server after each move. Further, the computational overhead of managing, disseminating, and maintaining consistency across views hinders server scalability [3], [4], [11]. A more scalable approach enables the server to perform *eager loading*, in which, move updates from one player are sent to every other player as they happen. Consequently, clients always have the game rendering information available locally. Since eager loading eliminates the burden of managing each player's view on the server and reduces game rendering delays, it is in widespread use by online games today. The disadvantage is that the game client now contains sensitive information belonging to other players that may not even be required to render the current game view, leaving open the potential for information exposure cheats.

In this paper, we present *SpotCheck*, a more balanced defense against information exposure cheats in client–server online games. SpotCheck enables the client, after each move, to compute and submit a description of its game view. The server then randomly chooses to validate the descriptor, if so, then a valid descriptor results in response containing the contents of the view. Like on-demand loading, SpotCheck eliminates the need for hidden information in the game client; and like eager loading it doesn't require the server to store and track a client's view. However, information exposure cheats can now occur by sending illegal view descriptors to the server. The advantage with SpotCheck is that the overhead of checking the view descriptor can be traded for improved game performance. For example, instead of checking every view descriptor, the server can randomly check only a fraction of them. Although this introduces opportunities for cheating, repeat offenders will eventually be identified. We study cheat detection time in Section VII.

The contributions of this paper are:

- The design and implementation of *SpotCheck*: an efficient and effective method for detecting information exposure cheats via random sampling and verification of a player's game state.
- An evaluation of SpotCheck's effectiveness and its resource requirements in the context of a toy multi-player real-time strategy game we call *Explorer*.
- A formal description of the class of games that can benefit from SpotCheck.

## II. Motivation

Our goal is to develop an effective and scalable solution for detecting information exposure cheats in client–server online games. Existing solutions are either effective and not scalable, or vice versa. Popular games like World of WarCraft [5] and StarCraft [6] use an anti-cheating software called Warden [7], which runs on the client, scanning periodically for signatures

of common cheats [8]. Tools like Warden are not always effective for two reasons. First, they cannot detect new information exposure cheats. For example, earlier map hacks were easy to detect because they were implemented by patching the game client, but new map hacks are harder to detect because they are external to the game and work by reading game memory and exposing the map areas via overlays [9]. Second, the scanning performed by such tools is widely considered to be a risk to personal privacy. The Electronic Frontier Foundation has even labeled Warden as "spyware" [10]. Another solution, called *on-demand loading* [11], proposes a change to the server's game state dissemination strategy that consequently prevents all forms of information exposure cheats. In this method, the server disseminates state information required to render the game only when clients need it. Thus, on-demand loading eliminates any hidden information in the game client, leaving no useful information to expose. Webb et al. [12] have called it the "most effective solution" against this form of cheating. Unfortunately, the price of on-demand loading is excessive server CPU overhead because the server must now track each client's view of the game. To avoid precisely this overhead, existing games like World of Warcraft prefer *eager-loading* [11]. Eager-loading involves sending state updates of one player to every other player. This eliminates the burden of tracking every player's view on the server and allows any client to independently render the game. However, now, each player has state information belonging to all other players whether or not it is needed to render their current view of the game. As a result, the extraneous information is kept hidden in the game client waiting to be exploited by those who know how to bypass cheat detection tools like Warden.

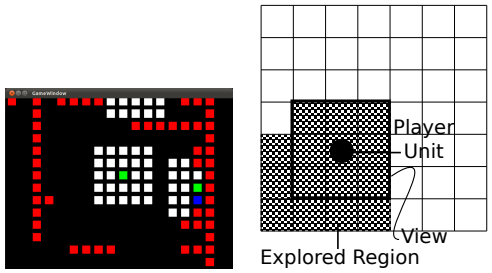
SpotCheck proposes a middle ground: trust the client to compute a description of its own view, but enable the server to verify the description's integrity. This approach does not require the server to track client views, instead it only requires the server to maintain global game state consistency and verify view descriptors. As long as the verification process is cheaper than tracking a client's view, our approach will be more scalable than on-demand loading. Additionally, like on-demand loading, game state information is disseminated to clients only as needed. Thus, there is no hidden information in the game client that cheaters can exploit.

There is, however, another approach that could be used to prevent information exposure cheats. Monch et al. [14] propose a technique that perpetually obfuscates all game state information stored in the client. The solution involves using decoding and encoding functions before accessing or writing any game state information. Also, since obfuscation does not provide cryptographic secrecy, the hiding functions are changed periodically via a secret channel. Thus, forcing an adversary to constantly reverse engineer the hiding functions.

If hiding functions are efficient and can indeed be surreptitiously changed faster than the time required to reverse engineer them, then the above solution may be effective. Unfortunately, the efficiency of the solution, the feasibility of constantly generating and secretly transmitting strong hiding functions, and in general the feasibility of developing tamper-proof software is still unclear [12].

## III. Overview

We analyze SpotCheck in the context of a simple real-time strategy game we call *Explorer*. The game consists of players exploring a 2D terrain consisting of walls and other obstructions (Figure 1(a)). Multiple players can play the game and each player can have multiple units exploring the terrain. The map of the terrain is composed of grids and each grid can contain either the terrain itself, part of a wall, a player's unit, or an obstruction. Additionally, unexplored regions of the terrain and those that are outside a unit's vision are kept hidden from the player (Figure 1(b)). At any given time, a unit's vision consists of the contents of a $5 \times 5$ grid around its current position. The only exception is that units cannot see through walls, thus, parts of the terrain that belong inside the vision but are blocked by walls will still be hidden. Player units can move one grid at a time in vertical or horizontal directions uncovering contents of unexplored regions of the terrain. In its current state, the game defines no objectives, player unit resources, or conditions for victory.




(a) Screenshot (b) Schematic

Fig. 1. Explorer overview

Formally though, SpotCheck is applicable to games consisting of the following high-level components:

- 1 to $n$ players
- 1 to $m$ units per player
- Global game state $S_t$: the state of the game on the server after move $t$, where $t$ is a global move counter across all players. The global game state for Explorer includes location and type information about terrain, obstructions, and player units.
- Game map $M$: a set of $e \times e$ square-shaped cell locations. We assume a square-shaped map and cell for simplicity, but SpotCheck need not be limited by the shape of the map or the cell.
- View descriptor $v_t^i$: the set of cell locations visible to all units of player $i$ after move $t$.
- View $V_t^i$: view descriptor $v_t^i$ along with game state information associated with cells in the descriptor.
- Explored Region $E_t^i$: game state information pertaining to the region of the map explored by all the units of player $i$ after move $t$.

- State request $U_t^i$: sent to the server by player $i$ after move $t$. The state request consists of the view descriptor $v_t^i$ ,unit identifier and the new location of the player's unit.
- State update $D_t^i$: sent from the server to the client after move $t$. The state update may contain a player's current view if generated in response to a state request, or it may contain changes to the player's current view as a result of moves by other players.

The game progresses as players make their moves and send corresponding state requests to the server. On receipt, the server verifies the integrity of the state request by randomly performing either a heuristic-based check, or a more expensive full check. State requests for illegal cell locations, could escape detection by heuristic-based checks, but will eventually be detected by the full check. We evaluate this further in Section VII. Once verification succeeds, the server sends back a state update. The game client then uses this state update to render a player's current view of the game. Intermittently, move updates from other players cause state updates to be sent to affected players. However, these updates only include changes to a player's current view.

Our game, Explorer, can be configured to disseminate state information using the three different strategies discussed earlier: on-demand loading, eager loading, and SpotCheck's strategy. The state request in on-demand and eager loading consists only of the player's new unit location, where as for SpotCheck, the state request contains the view descriptor as well. The state update in on-demand loading and SpotCheck consists of the player's view, where as in eager loading, it consists of information in a state request from any of the $n$ players.

## IV. Cheat Model

SpotCheck addresses application-level *information exposure* cheats [12], but unlike eager loading where these cheats are executed by accessing game client memory, SpotCheck forces cheaters to send malformed state requests to the server. Recall that players progress in the game by sending and receiving responses to state requests. These state requests contain the description of a player's view, which is verified by the server upon receipt. A cheater could conceivably construct a state request with an illegal view descriptor $v_t^{i'}$ that includes locations out of its actual view $v_t^i$. For example, by requesting state information for a cell across a wall, which by design blocks a unit's line of sight. If the server is unable to detect such malformed requests, then cheaters could potentially learn information about other players in any portion of the map.

In the context of SpotCheck, accessing game client memory is not considered an information exposure cheat. Mainly because SpotCheck eliminates the need for any hidden information in the game client. Note that hidden information is defined as any state information associated with map cells outside the explored region $E_t^i$. In the case where the player's view $V_t^i$ is a proper subset of the explored region $E_t^i$, this definition does allow game state information $E_t^i - V_t^i$ in the client that is not displayed to the player. However, this information is not considered hidden because it was first displayed when the player last visited those cells and has not been updated since.

Like on-demand loading, SpotCheck prevents infrastructure-level information exposure cheats by design. These type of cheats involve using a network hub and another host to sniff one's own game traffic and change the way it is rendered on screen. For example, modifying the display driver on the consorting host to render the game world with transparent walls. In SpotCheck, since information about cells outside a player's view (e.g. behind walls) is never included in state updates, it becomes futile to mount this type of cheat.

There are other ways of accessing hidden information that are outside SpotCheck's scope: a cheater might spoof a state request pretending to be another player. This type of cheat would need to be addressed by introducing appropriate authentication mechanisms [12]; a cheater might snoop another player's communication channel with the server. This type of cheat would need to be addressed by encrypting communications between the client and the server; a cheater might collude with other players to gain collective information; and lastly, a cheater might compromise the server to learn about global game state, which is normally not exposed to clients.

## V. Cheat Detection

The game server detects a cheat when it can't verify the integrity of a client's state request. State requests are sent to the server each time a client moves one of its units to a new location. The server then chooses to verify the state request with a probability $p$. The randomness ensures that a cheater does not know before hand if her state request is going to be checked or not. Without that knowledge, a cheater risks being detected and subsequently banned from the game. Since cheaters cannot progress in the game without sending state requests, they are forced to take their chances.

Notice that when $p = 1$, every state request is checked. The parameter $p$ essentially enables SpotCheck to provide game servers the flexibility of balancing resources used for game state management against those used for cheat detection. This is unlike on-demand loading, where the server is forced to track every client's view. The trade off with SpotCheck is that a cheater may initially get away, but with more cheat attempts a malicious player will eventually be identified. Thus, assuming that a player cheats during a move with probability $q$, the expected number of moves in which the cheater will be detected is:

$$E(T) = \sum_{t=1}^{\infty} t \times (1 - pq)^{t-1} \times pq = \frac{1 - pq}{pq} \qquad (1)$$

Where $T$ is a discrete random variable representing the number of moves a cheater can make before getting caught. We compare the expected outcome with the experimental result in Section VII.

The server validates a state request by performing a *full check* of its contents. A state request contains the player unit's new location (move), unit identity, and a view descriptor. A move is considered valid if the unit advances to an empty

and allowed map cell (currently, our game does not support multiple units in the same cell). A valid move results in an update to the global game state. Then, the updated game state information is used to construct the expected view descriptor for the player and compared to the one that was included in the request. Any discrepancies between the two is considered a cheat attempt. A state request that passes the full check is considered valid.

One disadvantage of probabilistically validating state requests is that easily preventable cheat attempts can occasionally escape detection. For example, a cheater may retrieve a snapshot of the global game state by constructing a view descriptor that includes all cell locations on the game map. This cheat can easily be prevented by validating the size of the view descriptor. To prevent such cheats, SpotCheck allows game servers to perform heuristic-based checks of the state request during those moves when the complete check described above is not performed.

We have currently implemented two heuristic-based checks of the state request. Each of these checks validate the range of legal values for individual components of the state request. The first heuristic, called *distance bound*, checks if the player's move is legal. So for example, in Explorer, units are not allowed to move more than one cell at a time. The second heuristic, called *vision size bound*, checks that the size of the view descriptor does not exceed the possible maximum for the player.

## VI. Architecture

The components of our game are split across the client and server. At a high level, the game client accepts key stroke input and renders the game, whereas the server validates the inputs and sends back the information necessary to render the game.

### A. Client Components

**Game map.** Stores the 2D game map. The map is composed of cells each of which contain a wall, a player unit, an obstruction, or the terrain itself. Information about the contents of the map is stored in the local game state.

**Game state manager.** All the information about a player's explored region is stored in the local game state. The state manager runs every time a move is generated and sends the move along with the current view descriptor in a state request to the server. The server validates the state request and sends back a state update containing information required to render the player's view.

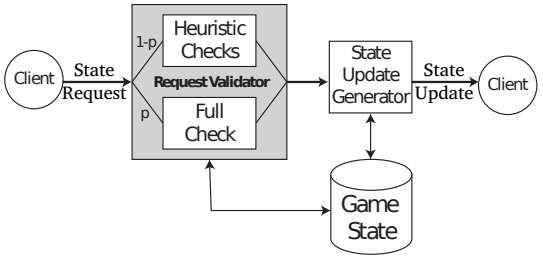
**Rendering engine.** Renders units, obstructions, walls, and terrain on visible portions of the map while blacking out the rest of the cells. After the state manager has received the necessary information required to render the player's view, it is passed on to the rendering engine, which then draws the view on the screen.

**Input mechanism.** Interprets key strokes as moves and forwards them to the game state manager.

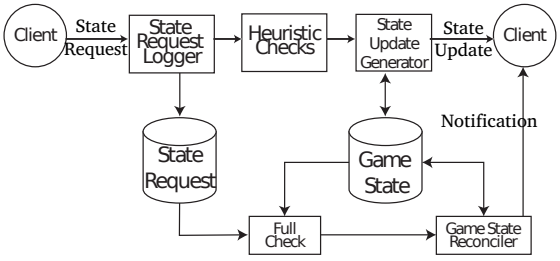
### B. Server Components

**Request validator.** As shown in Figure 2, the validator receives a state request from the client and chooses to either validate it with a probability $p$ or check it using simple heuristics (see Section V). A state request that passes the checks is then forwarded to the state update generator.

**State update generator.** Uses information in the state request to update global game state. Then, information pertaining to the player's view is retrieved from the global game state and sent as a state update.




(a) Current architecture




(b) Alternative architecture: request validation in parallel

Fig. 2. SpotCheck server-side architecture

It is worth mentioning that the state request validator need not run as part of the request processing pipeline (Figure 2(b)). The heuristic checks could still be done serially while the full check would happen in the background. Any game state corruption discovered during the full check may need to be reconciled. Although we have not explicitly evaluated this alternative, we believe that having the validator run in parallel can reduce state update response time and lead to an improved gaming experience with minimal impact on cheat detection integrity or performance.

Although SpotCheck is designed to detect information exposure cheats, it does not preclude including methods that detect other classes of cheats. For example, in the future, we plan to augment SpotCheck with code injection and entanglement algorithms [14] that prevent bots/reflex enhancer cheats.

## VII. Results

We will now evaluate SpotCheck against on-demand and eager loading while two players play a game of Explorer. The performance of eager loading forms our baseline since it provides no intrinsic protection against information exposure cheats and is currently the most prevalent method for disseminating game state information. On-demand loading is

very effective against protecting information exposure cheats, but is expensive for game servers to implement.

Our goal is to significantly reduce the overhead on the game server when compared to on-demand loading while providing similar levels of protection against information exposure cheats. Specifically, we will compare server CPU overhead, message sizes, and client game rendering latency of our approach with on-demand and eager loading. We will also evaluate the time it takes for SpotCheck to detect information exposure cheats.

### A. Experimental Setup

The test system used to gather performance data is a Dell Latitude E6510 configured with an Intel Core i5-M580 CPU running at 2.67 GHz with 4 GB of main memory. Intel TurboBoost and SpeedStep were disabled through the system BIOS. The operating system is a 32-bit Ubuntu 11.04 with Linux kernel version 2.6.38.9-generic. All experiments were performed with the game client and server on the same machine.

### B. Evaluation

Figure 3 plots server CPU overhead of the three schemes against units per player. Additionally, SpotCheck is plotted for the scenarios where 100% ($p = 1$), 25%, and 5% of the state requests are checked. The CPU overhead is measured as the total CPU time required by the server to process, check, and respond to a state request. Here, map size is held constant at $100 \times 100$ cells and the error lines indicate a 95% confidence interval over 200 runs per data point. Unless mentioned, the stated constants remain the same for all future plots.

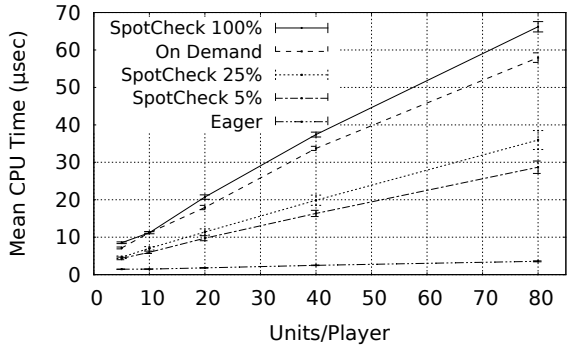



Fig. 3. Server CPU overhead

The CPU overhead of SpotCheck and on-demand grows because more units increase the size of a player's view, which consequently, requires more processing on the server. In eager loading, the game client deals with views, thus the server's overhead remains nearly constant and less than on-demand or SpotCheck. SpotCheck 100% incurs more overhead than on-demand loading because checking every state request is more expensive than tracking a player's view. This is not surprising because view tracking updates a stored view descriptor on the server with every state request, where as validation involves the more expensive operation of computing a new one from scratch each time (see Section V). Notice though that spot checking 5% of the requests requires nearly half the overhead of on-demand, where as checking 25% requires slightly more.

Figure 4 plots expected (Equation 1) and observed number of moves SpotCheck requires to detect an information exposure cheat versus various values of checking ($p$) and cheating ($q$) frequencies. A move where no cheating takes place is referred to as an "honest move", while the opposite is referred to as a "cheat move". For this experiment, we perform an information exposure cheat by issuing an illegal state request, where the view descriptor contains a location not in the player's current view. We also ensure that the information exposure cheat can bypass all heuristic checks. Thus, the cheat can only be detected by a full check. We can see that even when spot checking only 5% of the state requests, players cheating 50% of the time are detected before their $25^{th}$ cheat attempt. If players cheat less (5%) and SpotCheck checks less (5%) then the number of moves required to detect a cheater increase significantly, but the number cheat moves remain small ($\approx 15$).

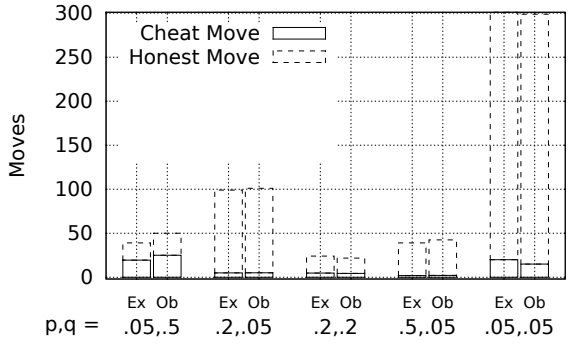



Fig. 4. Expected (Ex) and Observed (Ob) number of moves before a cheat attempt is discovered. Number of cheats that escaped detection are also shown as a portion of total moves.

Figure 5 (top) plots the size of a state update against the percentage of occupied locations visible to a player on the map. The size of the state update will significantly impact the server's outbound bandwidth requirements as players increase. Here, we focus on the request and update sizes for only one player with five units in the game. Additionally, 120 randomly chosen map locations are occupied by obstructions. In Explorer, the state update contains five bytes of information per location. We can see that under SpotCheck and on-demand, the size of the state update depends on the occupied locations visible to the player. Under eager loading, the server is oblivious to a player's view, thus it must send information about all the locations in question, forcing the update size to be larger than (or equal to) that of SpotCheck and on-demand.

Figure 5 (bottom) plots the size of the state request against the percentage of locations visible to a player on the map. Since the state request in eager and on-demand loading does not contain a view descriptor, its size remains smaller than

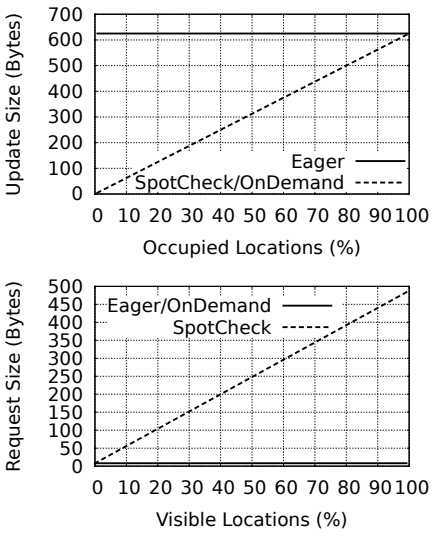



Fig. 5. State update and request message sizes

(or equal to) that of SpotCheck. Note, however, that Explorer uses the most naive method of encoding view descriptors: four bytes per visible location. Using better encoding schemes should result in much smaller request sizes.

Figure 6 plots client game rendering latency for SpotCheck ($p = 0.05$), on-demand, and eager loading. Rendering latency is measured as total CPU time between sending a state request and rendering the contents of the corresponding state update on screen. Keep in mind that the client and server are on the same machine hence the round-trip-time is fairly small. We can see that rendering latency increases with units per player. This is mainly because more resources are required to render the larger view of all units. SpotCheck and on-demand induce more latency than eager loading because of the additional time the server spends verifying the state request and tracking views respectively. Notice, however, that the latency incurred by SpotCheck is consistently less than on-demand even though SpotCheck requires more game processing on the client. This is because the server is more efficient under SpotCheck than under on-demand when responding to state requests.

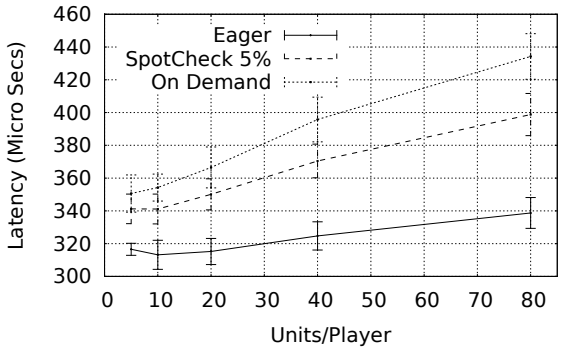



Fig. 6. Client game rendering latency for SpotCheck, on-demand, and eager loading

## VIII. Conclusion and Future Work

We presented SpotCheck, an efficient defense against information exposure cheats. SpotCheck strives to be the middle ground between on-demand loading, which prevents this class of cheats entirely, but is expensive for the game server to implement; and eager loading, which provides no protection from cheats, but is prevalent in online games today due to its performance benefits. The key idea behind SpotCheck is to allow the clients to track and request contents of their game view, but randomly sample and validate their requests. SpotCheck allows game servers to change the sampling frequency, thus trading off CPU overhead for increased cheat protection or vice versa. We have shown that SpotCheck can reduce the server CPU overhead by as much as half when compared to on-demand loading, while still being an effective defense against information exposure cheats. In the future, we plan to incorporate SpotCheck into a real-world game and evaluate its performs with a much larger number of players.

*This material is based upon work supported by the National Science Foundation under Grant No. CNS-1017034. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.*

## References

[1] S. Webb, S. and Soh, "A survey on network game cheats and P2P solutions," *Australian Journal of Intelligent Information*, vol. 9, no. 4, pp. 34–43, 2008. [Online]. Available: http://computing.edu.au/~webbsd/webb07e.pdf

[2] M. Pritchard, "How to Hurt the Hackers:The Scoop on Internet Cheating and How You Can Combat It," http://www.gamasutra.com/view/feature/3149/how_to_hurt_the_hackers_the_scoop_.php?page=3, July 2000, Gamasutra The Art & Business of Making Games.

[3] J. Vilches, "OnLive gets demoed, lag is a problem," http://www.techspot.com/news/37697-onlive-gets-demoed-lag-is-a-problem.html, Jan 2010.

[4] S. Miller II, "Beyond the hype of OnLive," http://www.jsonline.com/blogs/entertainment/41834997.html, Mar 2009.

[5] Blizzard Entertainment Inc., "World of Warcraft," http://us.battle.net/wow/en/.

[6] ——, "StarCraft II," http://us.battle.net/sc2/en/.

[7] M. Schramm, "Blizzard's new Warden, and our privacy," http://wow.joystiq.com/2007/11/15/blizzards-new-warden-and-our-privacy/, Nov 2007, WoW Insider.

[8] J. Thaler, "WardenNet," http://www.ismods.com/warden/wardenfaq.php.

[9] Artek72, "[Undetected] SC2MapPro - An External Map Hack/Bot," http://www.blizzhackers.cc/viewtopic.php?f=220&t=473310.

[10] Mark Ward, "Warcraft game maker in spying row," http://news.bbc.co.uk/2/hi/technology/4385050.stm, Oct 2005.

[11] K. Li, S. Ding, D. McCreary, and S. Webb, "Analysis of state exposure control to prevent cheating in online games," *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video - NOSSDAV '04*, p. 140, 2004. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1005847.1005878

[12] S. D. Webb and S. Soh, "Cheating in networked computer games," *Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts - DIMEA '07*, p. 105, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1306813.1306839

[13] H. Liu, M. Bowman, R. Adams, J. Hurliman, and D. Lake, "Scaling virtual worlds: Simulation requirements and challenges." in *Winter Simulation Conference'10*, 2010, pp. 778–790.

[14] C. Mönch, G. Grimen, and R. Midtstraum, "Protecting online games against cheating," *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games - NetGames '06*, p. 20, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1230040.1230087