SelectAudit: A Secure and Efficient Audit Framework for Networked Virtual Environments

Tuan Phan and Danfeng (Daphne) Yao

Department of Computer Science
Rutgers University, Piscataway, NJ 08854
{tphan,danfeng}@cs.rutgers.edu

Abstract. Networked virtual environments (NVE) refer to the category of distributed applications that allow a large number of distributed users to interact with one or more central servers in a virtual environment setting. Recent studies identify that malicious users may compromise the semantic integrity of NVE applications and violate the semantic rules of the virtual environments without being detected. In this paper, we propose an efficient audit protocol to detect violations of semantic integrity through a probabilistic checking mechanism done by a third-party audit server.

Keywords: networked virtual environments, algorithm, audit, integrity.

1 Introduction

Networked virtual environments (NVE) [10,11] refer to the category of distributed applications that allow a large number of distributed users to interact with one or more central servers in a virtual environment setting. For example, Second Life is a social NVE application [17]. Second Life is also called as a massive multiplayer online role-playing game, so is World of Warcraft [18] where a player assumes the role of a fictitious character in the game. Multiplayer online games enjoy great popularity around the world with the revenue exceeding one billion dollars in western countries [8]. For a conventional single-player local game, the graphics are rendered and simulated on the player's local machine. For multiplayer online games with a client-server model, when the number of players are small, it is still possible for the game server to centrally compute the graphics simulations and send them back to the players. However, for massively multiplayer online games, the main graphics renderings need to be done on the client's machines in order to reduce the workload of the game servers. Therefore, the server no longer has the entire control over what gets to be computed and displayed on the client's machine. Vulnerabilities and flaws in game designs are exploited by cheating players to unfairly take advantages of other players. Cheating behaviors discourage honest players from participating in the games [16] that hinders the development of game industry [5].

Multiplayer games can have two types of architectures: client-server or peer-to-peer. In most client-server models, a client sends to the server updates that affected the client's local state. The server then coordinates the global state

E. Bertino and J.B.D. Joshi (Eds.): CollaborateCom 2008, LNICST 10, pp. 207–216, 2009.

and adjusts the interactions between players. In comparison, a peer-to-peer is serverless [12] and relies on game players to coordinate their interactions and states among themselves. In this paper, we focus on the security of the client-server architecture.

The architecture of multiplayer games needs to be scalable to accommodate the interactions among a large number of players. In particular, the workload on the central server or a cluster of central servers need to be kept efficient to ensure responsiveness to clients (players)'s updates and requests. Because graphics rendering is computationally intensive, it is infeasible to make the central servers to perform the rendering for each player. Therefore, typical multiplayer online game servers only maintain abstract states of each player and the concrete outcome of simulation is performed and kept on the player's computer. The concrete state information captures the settings, contexts, environments, visual effects of the player at a given point. The abstract state can be thought of as a digest of the concrete state.

Both client-server and peer-to-peer architectures have potential vulnerabilities for cheating. As defined by Baughman *et al.* [1,2], cheating occurs when a player causes updates to game state that defy the rules of the game or gain an unfair advantage. For example, the game player may attempt to intercept and access hidden information, collude with his friend to learn secret information of his opponent [1], or lookahead cheat where a player simply waits until all other players have sent their decisions. Several cheating behaviors have been studied and solutions have been proposed by the research community, including lockstep protocols [1,4] for lookahead cheats, a secure online Bridge game design by [19], fair message ordering protocol [3], and an audit framework to prevent cheating on semantic rules [9]. From the game industry, cheating in multiplayer online games have been intensively discussed and studied [5,16].

Unlike most of the existing anti-cheat work, we study the semantic integrity of multiplayer games in the client-server architecture. Our goal is to develop a general and efficient audit framework to detect and deter this type of cheating. *Semantic integrity* of multiplayer online games is defined as that all the game players must observe and follow the logical rules that govern the simulation and interactions specified by the server.

The attacks on semantic integrity in the client-server architecture are due to several reasons. First, the client software can be modified by participants, which is easy for open-source games. Even for proprietary games, client modification is sometimes possible through reserve engineering. Therefore, a security solution should not assume that all clients are trusted. Second, due to the large scale of the game, the central server typically only keeps an abstract state of each player. Third, there is a difference between the central server's abstract state and the player's concrete state. For example, the central server may only store the coordination of a few points on a client's moving path, instead of the entire

[1] Certain online Bridge game allows one to join as a bystander and thus can view the cards of all players [19]. Therefore, a cheater can make his friend a bystander who can pass other players' cards information onto the cheater.

trajectory. This distinction is called *semantic gap* first by [9], as the state of a game player contains the semantic meanings. Recently, researchers have identified that semantic gaps may be exploited by malicious players to violate semantic rules of the game without being detected.

The main challenge in designing an audit framework for massively multiplayer online games is to prevent the audit server from becoming a performance bottleneck. Jha *et al.* proposed to use an audit server to catch cheating players by recomputing all of the audited players' previous game states. Their approach is simple and easy to implement. However, the heavy workload of the audit server is likely to create a bottleneck in the auditing process, as the recomputation of hundreds of thousands of game states is expensive. One easy mitigation is to deploy multiple audit servers to distribute the workload. Even with multiple audit servers, operations on each single auditor need to be optimized for efficiency as auditing needs to take place in real time to detect cheating players as quickly as possible.

In this paper, we develop a novel and efficient audit framework for multiplayer online games through the use of Merkle hash tree and a random verification mechanism. Our solution is designed for the client-server architecture. We propose a scalable algorithm that allows an audit server to periodically examine clients' game states to detect cheating events. The main feature of our solution is that under reasonable assumptions, an audit server only needs to recompute a constant number of game states of a client in order to catch a potential cheating client with a high probability. Therefore, the auditing protocol is scalable to hundreds of thousands of clients as typically seen in popular multi-player online games. (E.g., World of Warcraft currently has 8 million registered players.) Because of the use of Merkle tree [13,14], once caught, a cheating client cannot refute the evidences produced by the audit server.

2 An Example

Here, we show a simple example of semantic integrity violation. Table 1 illustrates the distribution of game information between the players and the central game servers.

Table 1. A table shows the distribution of game information between the players and the central game server

Type of an entity	Stores/computes	Where
Player	Concrete game state of the player	Player's local machine
Central game server	Abstract states of all players' states	Central game server

Example 1. To reduce storage requirements, the central game server may only keep the coordinates of the two end points of a moving player as part of the abstract state of the player, for example, in the right figure, point a at (x_1, y_1) and point b at (x_2, y_2). As a result, a cheating player may violate the game rules by

walking through walls (the red path) without being detected by the state server! A correct path is to follow the green path. Without an secure audit mechanism, the central game server is unable to detect this type of cheating events.

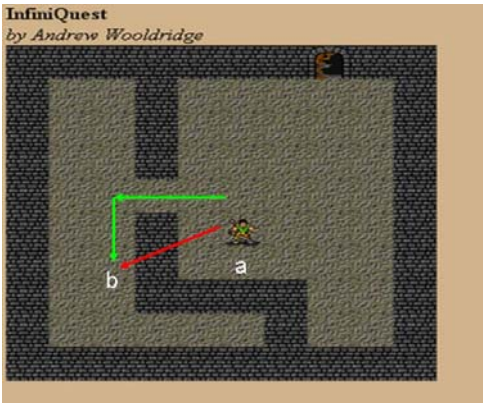

Fig. 1. Walking through walls: an example of semantic integrity violation. The central game server only keeps the coordinates of the two end points of a moving player as part of the abstract state, point a at (x_1, y_1) and point b at (x_2, y_2). A cheating player may violate the game rules by walking through walls (the red path) without being detected by the state server.

Other possible semantic integrity attacks include seeing hidden objects, shooting from the back, and reversing explosion damages, just to name a few. In some cases, semantic integrity violation is a result of other types of attacks such as reflex augmentation. For example, shooting from one's back (without seeing the target) is due to the use of aiming bot that is a program that automatically shoots once the target position is obtained (mostly by examining network packets). Therefore, catching semantic violations may also reduce other attacks in multi-player online games.

How to define semantic integrity rules that are to be enforced is specific to an application, which is out of the scope of this paper. However, open questions remain as to the complexity of such set of rules and how to easily generate these semantic rules by designing automatic tools. Our approach to detect semantic violations is to use an audit server to audit players by selectively recomputing and verifying players' concrete states, which is presented in the next Section. In what follows, we assume that the audit server has already obtained a set of semantic rules that it needs to enforce.

3 Preliminaries

In this section, we introduce the preliminary knowledge needed to understand our protocol. We briefly explains Merkle hash tree and the necessary cryptographic

primitives that are used by us. We also briefly describe a simple audit approach that will be compared to our protocol.

We use Merkle hash trees for authentication of values $a_1, \ldots, a_n$. Merkle hash tree is for efficient authentication of a large number of items. This simple and elegant data structure has previously been used in various occasions [6,7]. A binary Merkle hash tree is a tree where an internal node h_0 is computed as the hash value $H(h_1, h_2)$ of two child nodes h_1 and h_2. In our construction, the order of inputs in the hash function matters and represents the node position in the tree, e.g., h_1 is the left node. The root hash y of the tree represents the digest of all the values at the leaf nodes, which are values $a_1, \ldots, a_n$. To authenticate that leaf a_i is in the hash tree, the proof is a sequence of hash values corresponding to the siblings of nodes that are on the path from a_i to the root. To verify the proof, anyone can recompute the root hash with a_i and the sequence of hash values in the proof. In our SelectAudit protocol, the Merkle tree can be thought of as the commitment on the game information by a player. Given a leaf node on a Merkle hash tree, the *proof nodes* refer to the minimum set of nodes that are required to construct the root hash. In other words, proof nodes consist of the sibling nodes of the leaf node on the path from the root to the leaf node.

The root hash of the Merkel tree needs to be authenticated with a digital signature using a public key signature scheme or a keyed-hash message authentication code (HMAC) with a shared secret key between the signer and the verifier. For online game settings, public key signature scheme is not suitable as a player may not possess a public and private key pair. Therefore, a shared secret session key is usually generated between the player and the central server, then the player uses the shared key to create HMAC on the root hash for authentication purpose. Our protocol assumes the existence of a collusion-resistance one-way hash function that (1) it is hard to compute the input of the hash function from the hash value and (2) it is hard to find two distinct messages that give the same hash values.

For the ease of discussion, we refer to the audit protocol presented in [9] as the SimpleAudit protocol. In SimpleAudit, each audited player has to compute HMAC on *all* of the update messages and send them to the auditor for verification. The auditor detects violation by performing the following three main operations.

1. To verify the MAC of each update message to ensure message authenticity.
2. To render *each* concrete game state corresponding to each update.
3. To verify the compliance of each concrete state according to the semantic rules of the game.

We describe our protocol SelectAudit in the next section.

4 Our Approach

In this section, we show how to apply cryptographic tools to prevent semantic integrity violation attacks in NVE. Our audit protocol reduces the computation costs at the audit server and the communication costs between the client and the

audit server by using Merkle hash trees. Our aim is to improve the performance of the audit server so that it can efficiently audit a large number of players simultaneously.

Table 2. Notations in our SelectAudit protocol

k	Shared secret key between Client and AuditServer
HMAC(k, M)	HMAC of a message M by using a key k
S_0	Beginning concrete state of an audit cycle
Q_0	HMAC on S_0
S_c^t	Concrete state at the time t in the audit cycle c
Δ_i	Update on Client's concrete game state at epoch i
δ_i	Abstraction corresponding to Δ_i (i.e., update on abstract state)
Root hash	Root hash of Client's Merkle hash tree
Q_c	HMAC of root hash
$\hat{S}_i$	Concrete state of Client at epoch i recomputed by AuditServer

4.1 Overview

There are three types of entities in our protocol: Client, StateServer, and AuditServer. StateServer is the central state server that coordinates the players and maintains the abstract states of all the players. AuditServer does auditing on all the players. Client refers to a player. AuditServer and StateServer are mutually trusted by each other. Client is not assumed to be trusted by the servers. To avoid requiring AuditServer to recompute *every* concrete state corresponding to an update of an audited client, we design a sampling technique. AuditServer only checks the semantic integrity associated with m updates out of n updates in an auditing cycle. By choosing a reasonably large m, a cheating player can be detected with high probability. We call our audit protocol *SelectAudit*.

An audit cycle refers to a time period specified by the game, during which a client's game records are examined by the audit server. The audit cycle is agreed upon by all the entities in the game system. An audit cycle consists of n number of epoches, each is associated with an update on the client's game state. During each cycle, each client constructs a Merkle hash tree on the update messages (no matter he is under audit or not). At the end of the cycle, the client computes the root hash of the tree and a message authentication code on the root hash. The client saves these values. If at a later point of time, the audit server decides to audit the client for a previous cycle, part of the stored Merkle tree corresponding to that cycle is sent to the auditor for verification. Without loss of generality, we choose n to be power of 2 for the ease of building the Merkle tree by the client.

For example, suppose the audit cycle consists of 8 epoches. A client joins the game at epoch 1. At each epoch from 1 to 8, he constructs the Merkle hash tree incrementally based on his game state information, and computes a MAC on the root hash at the end of epoch 8. At epoch 10, the audit server notifies the client that he is under audit. The client then engages in the audit protocol using the Merkle tree that he previous generated.

4.2 Our SelectAudit Protocol

Our SelectAudit protocol has three components: INITIALIZE, STATEUPDATE, and AUDIT, each of them is a protocol itself that is run. In what follows, we assume that the StateServer and AuditServer have a secure channel for communicating messages.

Initialize: This protocol is run among the StateServer, the AuditServer, and the Client when the Client first joins the online game. The StateServer sends the initial *concrete state* for the client based on the client's profile. As it is chosen by the StateServer, this initial concrete state of the client satisfies the semantic integrity of the game. The client also commits to AuditServer on the initial state by sending it a HMAC of the initial state. ASTATE is a shorthand for abstract state.

1. Client and AuditServer exchange a secret session key k, that is used to generate HMAC values in the updating phases and the auditing phases.
2. Client initializes $t = 0$ and sends an initialization request to StateServer.
3. StateServer chooses a concrete state S_0 for the client based on his profile. StateServer sends to the client the initial state S_0.
4. Client computes and stores $Q_0 = \text{HMAC}(k, S_0)$ along with the concrete state S_0 for audit purpose.

StateUpdate: This protocol is mainly run by the Client and StateServer to compute an updated game state for each epoch t of the game. The client also maintains the Merkle hash tree in case he gets audited later on. The Merkle hash tree is built on top of updates $\{\Delta_i\}$. The Merkle tree can be thought of as the commitment on the updates by the Client. For each epoch t in an audit cycle c,

1. Client computes a desired status update Δ_{t+1} and its corresponding abstraction δ_{t+1}.
2. Client sends δ_{t+1} to the StateServer.
3. Upon receiving δ_{t+1}, StateServer computes a new δ'_{t+1} and updates its abstract state accordingly.
4. StateServer sends the following to both Client and AuditServer: $(\delta'_{t+1} \parallel t+1 \parallel Client_{id})$.
5. Client chooses and stores the concrete update $\Delta'_{t+1} \in \gamma(S^t_c, \delta'_{t+1})$ and computes the new concrete state $S^{t+1}_c = S^t_c + \Delta'_{t+1}$.
6. Client inserts Δ'_{t+1} into the Merkle tree corresponding to the current audit cycle.
7. Client increments t. At the end of cycle c, i.e., $t == n$ where n is the number of epoches in an audit cycle,
   (a) Client computes the corresponding new concrete state S^c.
   (b) Client computes $Q_c :=$ HMAC (k, root hash), then sends Q_c to AuditServer. Note that this step is required for each Client for each cycle.
   (c) Client initializes for the next audit cycle by setting $t := 0$ and beginning concrete state $S_0 := S^n_c$. Client computes and stores HMAC(k, S_0) along with the beginning concrete state S_0 for audit purpose.

Audit: To audit a previous cycle c on Client, AuditServer and Client engage in a protocol that allows the AuditServer to verify that (1) game renderings based on the beginning concrete state S_0 and updates at *selective* epoches satisfy semantic integrity, (2) the beginning concrete state and updates submitted by Client are authentic.

1. AuditServer informs Client that he is under audit for an earlier cycle c.
2. Client sends to AuditServer all the concrete updates $\{\Delta'_i\}$ in the audit cycle for all $i \in [1, n]$, and the beginning concrete state S_0 of cycle c, and its HMAC Q_0.
3. AuditServer checks whether VerifyHMAC(k, root hash, Q_c) = TRUE and VerifyHMAC(k, S_0, Q_0) = TRUE. These two steps are to verify the authenticity of the root hash of Merkle tree and the beginning concrete state S_0 that are received from Client. Recall that Q_c and root hash are sent by Client to AuditServer in Protocol StateUpdate.
4. AuditServer randomly picks m numbers from $[1, n]$ that represent the indices of epoches to be audited in audit cycle c. These numbers form the challenges for Client and are denoted by *challenge_set*. AuditServer sends the *challenge_set* to Client.
5. Client prepares a response message M that includes the proof nodes on Merkle tree corresponding to Δ'_j where $j \in$ *challenge_set*. Recall that proof nodes defined in Section 3 consist of the sibling nodes of the leaf node on the path from the root to the leaf node. Client sends to AuditServer: $M \parallel \text{HMAC}(k, M)$.
6. AuditServer verifies the HMAC on M. Then, for each challenge $i \in$ *challenge_set*:
   (a) Auditor verifies the authentication of Δ'_i by reconstructing the root hash of the Merkle Tree from Δ'_i and its proof nodes. If the reconstructed root hash should be the same is the one sent in StateUpdate.
   (b) Auditor re-computes the concrete state of Client associate with epoch i by $\hat{S}_i = S_0 + (\Delta'_1 + \Delta'_2 + ... + \Delta'_i)$, i.e., to compute the concrete state by applying accumulated updates.
   (c) Auditor checks whether Δ'_i chosen from $\gamma(\hat{S}_i, \delta'_i)$ is compliant with the rules of the game. How to define the rules of game depend on the specific NVE application and is out of the scope of this paper. Recall that δ'_i is obtained from StateServer in Step 4 in Protocol StateUpdate.
7. AuditServer accepts the computation of Client if and only all the above tests pass. Client may delete the stored audit records.

AuditServer needs to maintain the auditing schedule of each client, i.e., when to audit which subset of clients. The process of choosing which subset of clients to audit should be randomized as opposed to following a predictable pattern. Otherwise, a cheating client can predict the cycles that he will be audited and cheats for the rest of the time. Ideally, for each audit cycle, every client is audited, which gives the best guarantee on detecting semantic integrity violations. However, this type of scheduling gives the audit server a heavy workload. Thus, there is a tradeoff between efficiency and detectability.

Theorem 1. *Assuming the existence of collision-resistance one-way hash function, our SelectAudit protocol preserves the semantic integrity of NVE and is secure against probabilistic polynomial-time adversaries in NVEs in the following attacks: message tampering and forgery, audit replay attacks, refuting audit results attacks, collision attacks, reordering attacks, and tailered update attacks.*

Theorem 2. *Let n be the number of updates in an audit cycle, m be the number of updates the audit server challenges a player, and r be the honest ratio that is defined as the probability that a player does not cheat. Also let ϵ be the maximum allowed probability of cheating without being caught. Then in SelectAudit, the following formula captures the upper bound on m.*

$$m < \frac{\log \epsilon}{\log r}$$

Due to space limit, we omit the detailed security and performance analysis. We refer readers to the full version of our paper for more information [15].

Comparison with SimpleAudit. Jha *et al.* conducted the first study on the semantic integrity of multi-player online games [9]. The SimpleAudit protocol in Section 3 captures the essence of their protocol. Our SelectAudit improves SimpleAudit in terms of both audit server efficiency and communication overhead. In [9], when a player is audited for a previous time period t, he needs to send all of the game updates and their message authentication code (MAC) associated with t to the audit server. In our solution, the MAC is only computed *once* for time period t. This simplification not only saves the communication overhead between the players and the audit server, but also lowers the computation overhead for the players. In [9], the audit server needs to recompute all of the concrete game states of each player who is under audit. Because game rendering is expensive, this recomputation is a significant overhead, especially when the number of players to be audited is large. In comparison, our audit server only needs to recompute a selective number of concrete states for each player, in order to have a high detection probability.

5 Conclusions

We have described a general and scalable audit framework for massive multiplayer online games and for networked virtual environments in general. We are able to develop a distributed and efficient audit protocol that is run by the audit server to periodically examine the semantic integrity of clients' game states. The audit server is able to quickly detect cheating players and provide irrefutable proofs on the cheating behavior. The main advantage of our random checking algorithm is that the audit server only needs to perform a small number of rendering operations in order to catch cheaters with a high probability. This randomization significantly saves the computation costs for the audit server.

References

1. Baughman, N.E., Levine, B.N.: Cheat-proof playout for centralized and distributed online games. In: IEEE INFOCOM, pp. 104–113 (2001)
2. Baughman, N.E., Liberatore, M., Levine, B.N.: Cheat-proof playout for centralized and peer-to-peer gaming. IEEE/ACM Trans. Netw. 15(1), 1–13 (2007)
3. Chen, B.D., Maheswaran, M.: A cheat controlled protocol for centralized online multiplayer games. In: Proceedings of the 3rd Workshop on Network and System Support for Games (NETGAMES), pp. 139–143. ACM Press, New York (2004)
4. Chen, B.D., Maheswaran, M.: A fair synchronization protocol with cheat proofing for decentralized online multiplayer games. In: 3rd IEEE International Symposium on Network Computing and Applications (NCA), pp. 372–375 (2004)
5. Davis, S.B.: Why cheating matters. In: Game Developer's Conference (2003), http://www.secureplay.com/papers/docs/WhyCheatingMatters.pdf
6. Du, W., Jia, J., Mangal, M., Murugesan, M.: Uncheatable grid computing. In: International Conference on Distributed Computing Systems (ICDCS), pp. 4–11
7. Haber, S., Hatano, Y., Honda, Y., Horne, W., Miyazaki, K., Sander, T., Tezoku, S., Yao, D.: Efficient signature schemes supporting redaction, pseudonymization, and data deidentification. In: Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security (ASIACCS), pp. 353–362 (2008)
8. Harding-Rolls, P.: Western world MMOG market: 2006 review and forecasts to 2011, Management Report. Screen Digest (March 2007)
9. Jha, S., Katzenbeisser, S., Schallhart, C., Veith, H., Chenney, S.: Enforcing semantic integrity on untrusted clients in networked virtual environments. In: IEEE Symposium on Security and Privacy, pp. 179–186 (2007)
10. Joslin, C., Giacomo, T.D., Magnenat-Thalmann, N.: Collaborative virtual environments: From birth to standardization. IEEE Communications Magzine, 28–33 (April 2004)
11. Joslin, C., Pandzic, I.S., Magnenat-Thalmann, N.: Trends in networked collaborative virtual environments. Computer Communications 26(5), 430–437 (2003)
12. Knutsson, B., Lu, H., Xu, W., Hopkins, B.: Peer-to-peer support for massively multiplayer games. In: IEEE INFOCOM (2004)
13. Merkle, R.: Protocols for public key cryptosystems. In: Proceedings of the 1980 Symposium on Security and Privacy, pp. 122–133. IEEE Computer Society Press, Los Alamitos (1980)
14. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990)
15. Phan, T., Yao, D.: Select audit: A secure and efficient audit framework for networked virtual environments. Technical Report DCS-TR-642, Rutgers University (2008)
16. Pritchard, M.: How to hurt the hackers: The scoop on internet cheating and how you can combat it, http://www.gamasutra.com/features/20000724/pritchard_pfv.htm
17. Second Life, http://secondlife.com/
18. World of Warcraft, http://www.worldofwarcraft.com/index.xml
19. Yan, J.: Security design in online games. In: ACSAC 2003, Washington, DC, USA, p. 286. IEEE Computer Society, Los Alamitos (2003)