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, e.g., Second Life and multi-player online games. 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, e.g., going through a wall. This problem is partly due to the fact that the central server only maintains an abstract information of a player's state. In this paper, we propose an efficient audit framework to detect violations of semantic integrity through a probabilistic checking mechanism done by a third-party audit server. We present a secure audit protocol that periodically requests a NVE client to submit his or her state information to an audit server who then performs selective recomputation to verify the client's results. Because typically NVE has a large number of players, an audit protocol needs to be scalable.

Key words: 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 massively multiplayer online role-playing game, so is World of Warcraft [19] 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 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 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 rending is computation 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 step 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 gains 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 [20], 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 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 multi-player 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 multi-player 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.

Our solution has general applications in the security of distributed systems, which is beyond the specific multi-player online game problem studied.

The rest of the paper is organized as follows. We describe an example of semantic integrity violation in Section 2. We give the preliminary knowledge in Section 3. Our audit protocol is presented in Section 4. We analyze the security and performance of our protocol in Section 5. Related work is described in Section 6. Conclusions and future work are given in Section 7.

[1] Certain online Bridge game allows one to join as a bystander and thus can view the cards of all players [20]. Therefore, a cheater can make his friend a bystander who can pass other players' cards information onto the cheater.

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.

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

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

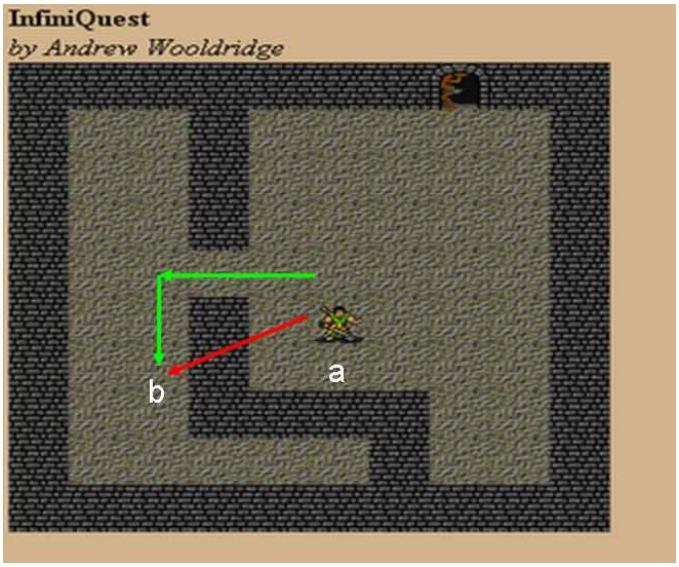


Fig. 1. Walking through walls: an example of semantic integrity violation. The central game server only keeps the coordinates of the two end points of a moving player as part of the abstract state, point a at (x_1, y_1) and point b at (x_2, y_2). A cheating player may violate the game rules by walking 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 (See also the related work Section). 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 wants 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 a simple audit approach that will be later compared to our protocol in Section 5.

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.

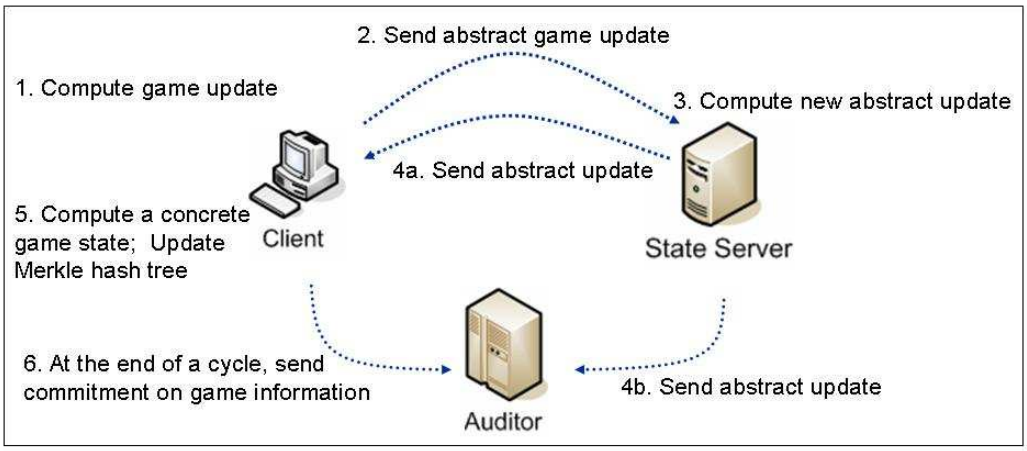
The above SimpleAudit protocol is analyzed further in Section 5. We describe our protocol SelectAudit in the next section.

4 Our Approach

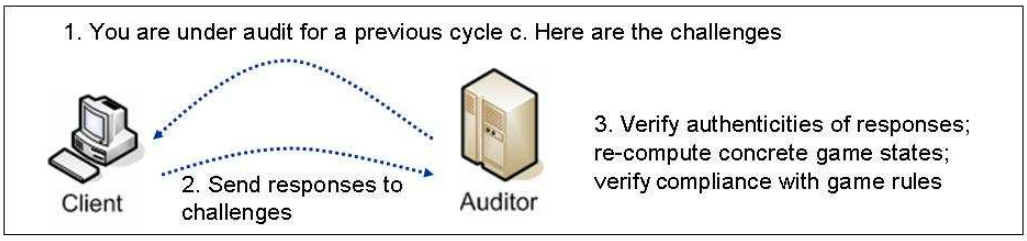
In this section, we show how to apply cryptographic tools to prevent semantic integrity violation attacks in NVE. Our audit protocol reduces the computation costs at the audit server and the communication costs between the client and the audit server by using Merkle hash trees. Our aim is to improve the performance of the audit server so that it can efficiently audit a large number of players simultaneously.

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 of an audit cycle recomputed by AuditServer

Table 2. Notations in our SelectAudit protocol.



(a) Schematic drawing of protocol StateUpdate



(b) Schematic drawing of protocol Audit

Fig. 2. Schematic drawing of Protocol StateUpdate in (a) and Protocol Audit in (b).

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. In Section 5, we show that if we choose for a reasonably large m, the cheating is almost impossible. 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 $\{\Delta\}$. 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,
  1. Client computes the corresponding new concrete state S^c.
  2. Client starts to compute $Q_c :=$ HMAC (k, root hash), then Client sends Q_c to AuditServer. Note that this step is required for each Client for each cycle.
  3. 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 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 the 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 concrete updates in the audit cycle $\{\Delta'_i\}$ 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 in 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.

5 Analysis

In this section we analyze the security properties of our SelectAudit protocol, and give performance results on the performance.

5.1 Security Analysis

We first analyze several types of attacks that malicious players may attempt to launch on SelectAudit protocol. Then we summarize the security guarantees provided by SelectAudit. Our protocol is secure against the conventional network attack of packet tampering and forgery, because of the use of message authentication code and the fact that random keys are hard to guess. Therefore, a receiver can verify the authenticity and the integrity of the messages. Encryption scheme can be applied easily to SelectAudit to achieve message confidentiality that is against packet eavesdropping. In the following, we discuss attacks that are specific to auditing and explain how our protocol is secure against them.

Audit replay attack We define audit replay attacks as a malicious player sends previously recorded correct state information to the audit server, instead of sending the actual game state of the audit cycle that

may be violating semantic integrity. Our protocol is secure against replay attacks because in StateUpdate, the state server sends to the audit server (and also the player) δs for abstract updates. Audit server uses these δ values in Audit phase to verify semantic integrity. Therefore, a previously recorded state information does not match and replay attacks can be identified.

Refuting audit results Our protocol provides accountability on the audit information. A cheating player, once caught, is held accountable for his unfair game play. Upon dispute, the audit server is able to produce irrefutable evidences to prove that a player violates the game rules. This property is possible because the player commits by creating HMACs on the game states and update messages. As the secret key for HMAC is only accessible by the player, he cannot deny computing these illegal game states.

Collision attack We define collision attacks as follows. An attacker attempts to find two different concrete updates Δ_1 and Δ_2 that give the same hash value. If the attacker successes, then he can cheat in Δ_1 (i.e., Δ_1 has illegal moves that violate semantic integrity of the game), but report a legitimate update Δ_2 instead. Because of our assumption on the existence of collision-resistance one-way hash function, a cheating player is unable to carry out collision attacks successfully. The above definition on collision attack gives the adversary the freedom to choose both updates. In reality, what an attacker really needs is to find an Δ_2 that collides with a given Δ_1, which is even harder to find [18].

Reordering attack If an attacker learns the *challenge_set*, he may attempt to reorder and swap the updates Δs. The audited update at epoch i ($i \in$ *challenge_set*) may be swapped with a legitimate update at $j \notin$ *challenge_set*). Reordering attacks are prevented in SelectAudit due to two reasons. (1) Recall that in our Merkle hash tree construction, the order of inputs in the hash function matters and represent the node position in the tree, e.g., h_1 is the left node. Thus, reordering can be detected when Δ_i is verified. (2) In SelectAudit, a player commits to all the updates Δs, before a player learns the *challenge_set*. He has to send and commit to all the concrete updates $\{\Delta_i\}$ within an audit cycle to the audit server first. Thus, the attacker does not know which set of epoches are to be audited.

Tailered update attack Tailered update is a subtle attack that has a similar flavor as the reordering attack. Suppose that an attacker violates the semantic integrity in Δ_i. If an attacker learns that epoch i is to be audited whereas $i-1$ is not, he may attempt to modify the update Δ_{i-1} so that the resulting concrete state permits all the updates in Δ_i. In SelectAudit, this attack is impossible to carry out because *challenge_set*, he has to commit all the updates $\{\Delta_i\}$ first before learning *challenge_set*.

To summarize the above discussion, we give Theorem 1.

Theorem 1. *Assuming the existence of cryptographic 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.*

We omit the formal proof of Theorem 1 here as it can be easily derived from our analysis of the attacks. In SelectAudit, AuditServer obtains the beginning concrete state S_0 of Client for audit cycle c from Client. As shown in Audit protocol, S_0 is the base to verify semantic integrity. Suppose that S_0 is an "illegal" state that is not computed following the game rules. Also suppose that subsequent updates are done following the game semantic rules. Then the client's previous cheating act will not be detected by the audit server. This issue can definitely be solved by having the audit server to run frequent audit checking on a player. The audit server ensures that the last state of an audit cycle (that is also the beginning state of the next audit cycle) always satisfies semantic integrity.

We analyze the probability of a client passing the audit successfully, following [6]. Denote n as the number of updates in one audit cycle. Denote m as the number of updates the audit server challenges the client. Denote r as the honest ratio that is defined as the probability that a client does not cheat. Also let ϵ be the maximum allowed probability of cheating without being caught. We want to find a upper bound on m that satisfies ϵ.

For the ease of analysis, we assume that any two updates at different epoches are independent of each other. Thus, if a player only violates the semantic integrity at epoch i, he can only be caught if the audit selects epoch i to verify his game state. The player will not be detected if the audit server audit a later epoch $j > i$, or a previous epoch $j < i$.

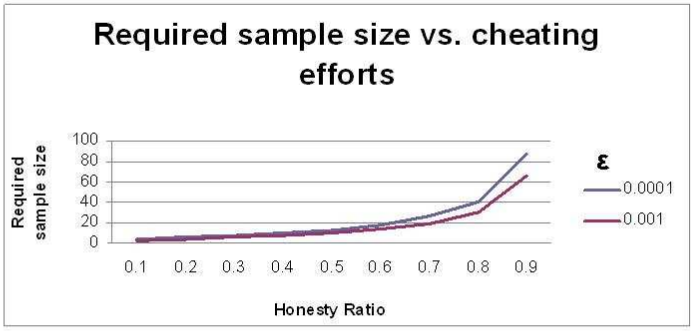


Fig. 3. The relationship between the required number of challenges with the honest ratio of a player, given ϵ, 0.001 (lower line) and 0.0001 (upper line), which is the maximum tolerated probability of cheating without being caught.

$$P_1 = Pr[\text{cheating without being caught}] = r^m$$

To bound P_1 under ϵ, we have

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

In Figure 3, we show the relationships between m and r under different ϵ. The analysis shows that the audit server only needs to audit a reasonably small number of epoches to ensure a low cheating rate, for a small ϵ. As ϵ gets smaller, m goes up. In the extreme case where ϵ is zero, then m needs to be the same as n. In other words, if the game system has zero tolerance on cheating, then the audit server needs to recompute the players' states for every single epoch.

Suppose that if a cheating event at epoch i may be detected at a later epoch $j \geq i$, then the audit server will have a higher chance of detecting a cheating player. We do not give the analysis on this case, as this may not be the case for most online games.

Theorem 2. *Let n be the number of updates in one 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.*

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

5.2 Performance Analysis

In SimpleAudit protocol defined in Section 3 [9], clients have to create HMACs for all differences and send HMACs to auditor. This security mechanism requires $O(n)$ communications between clients and the audit server, where n is the total number of epoches in an audit cycle. The computation cost at the audit server to verify HMAC is $O(n)$, which is to verify each update Δ_i for $i \in [1, n]$. In comparison, in our SelectAudit, we use the Merkle Hash tree to allow all n differences to be committed efficiently and the verification of each single difference to be performed efficiently. For each audit cycle, the audit server verifies one HMAC,

which is the HMAC on the root hash, and $O(m \log n)$ hash operations where m is the number of challenges randomly selected by the audit server. The communication costs between the client and the audit server still $O(n + m \log n)$ as we require the client to send all the concrete updates to the audit server and proof nodes of challenges. We compare the operation overhead in SimpleAudit and SelectAudit in Table 3.

Entity	Operations	SelectAudit	SimpleAudit
Audit server	Hash	$O(m \log n)$	$O(n)$
Audit server	HMAC	$O(1)$	$O(n)$
Audit server	Rendering concrete states	$O(m)$	$O(n)$
Audit server	Communication with player	$O(n + m \log n)$	$O(n)$
Player	Hash	$O(n)$	$O(n)$
Player	HMAC	$O(1)$	$O(n)$
Player	Communication	$O(n + m \log n)$	$O(n)$
Player	Storage	$O(n)$	$O(n)$

Table 3. Performance comparisons between SimpleAudit and SelectAudit. Let n be the number of updates in one audit cycle, m be the number of updates the audit server challenges a player. The table shows the complexity for an audit server to audit one player.

6 Related Work

Besides semantic attacks, there are several types of attacks on multiplayer online games that the game developing community has discussed. Some of the preventions are quite straightforward or have flaws. Nevertheless, we mention them here to give the readers a bigger picture of the security issues involved in online games. As the security of online games has not been systematically studied by the research community, some of the interesting problems do not yet have satisfactory solutions. Pritchard gave an excellent overview on various cheating behaviors in computer games in 2000 and also talked about the limitations of various game architectures as they relate to multiplayer cheating [16]. Reflex augmentation refers to an attack that uses a computer program to replace human reaction to produce superior results [16]. For example, an aiming bot is such a reflex augmentation. To detect reflex augmentation, the game server may use a simple comparison approach that compares the reaction time and accuracy or a player with statistical information (e.g., average response time of all players). However, a smart attacker may perturb its reflex augmentation so that it does not always achieve perfect results to prevent being detected. False positive is another problem associated with this type of detection where a truly good and honest player is misidentified as a cheater. Overall, this type of attacks is hard to detect.

Authoritative client is defined by Pritchard to refer to a type of attacks where a malicious player sends false updates to other players to mislead others, e.g., "player 2 has 10,000 hit points" [16]. Authoritative client attacks happen mostly in peer-to-peer online games where there is no central game server. In client-server settings, this attack means that a malicious player places false updates to other players on behalf of the server. This type of authoritative client attacks can be easily prevented by using authentication mechanism so that a player only accepts updates from the authenticated server. In P2P environments, one solution is to carefully engineer a synchronization check that is to be performed by the players in a collaborative fashion [16]. The security model, assumption, and guarantee of this approach have not been studied.

Information exposure is a type of attacks where players attempt to intercept, analyze and take advantage of hidden information, for example, (1) trying to see the objects hidden behind walls or (2) accessing data sent to other players by intercepting network packets. The use of encryption schemes can be used to achieve data confidentiality of client-server communication that foils eavesdropping attacks. The first type of information exposure (i.e., seeing through walls) is essentially a semantic violation. One approach to prevent players

from assessing hidden data in the memory is to use simple encryption scheme that randomize the value while storing it in memory. One technique against information exposure that was briefly mentioned in [16] is related to enforcing semantic integrity of multiplayer games. To combat hacked game view, e.g., seeing through hidden objects, statistical information on what gets drawn to the screen can be collected from each player. The statistics are derived from the game simulation data (e.g., on whether or not a player can see the object he just clicked on). The statistics are sent to other players in the game and abnormal statistics indicating possible cheating behaviors may be identified by the community. Essentially, the statistical approach is a collaborative way for an auditor to check semantic integrity of games which gives an approximation on how honest a player is in comparison to others. However, this method crucially depends on that the community is honest majority and most players possess the *correct* statistics. Thus it may be vulnerable to collusion attacks or attacks where cheating players are majority.

Jha *et al.* gave the first study on the semantic integrity of multi-play online games [9]. They proposed an audit protocol that is less efficient than ours in terms of both audit server performance and communication overhead. In [9], when a player is audited for a previous time period t, he needs to send all of 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 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 significant overhead, especially 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 high detection probability.

Trusted computing module (TPM), which is designed for digital rights management, can be used to reduce cheating events in online games. It requires players to use TPM to perform remote attestation that demonstrates the authenticity of the unmodified code on the player's local machine [15]. In comparison, our solution does not rely on the use of TPM, as we assume that the players are not trusted. We would also like to point it out that in certain cases such as P2P games, the use of TPM may not be feasible, as it requires a central server to perform the attestation. TPM cannot eliminates all of semantic integrity attacks because a cheater may leverage the use of abstract state to exploit semantic vulnerabilities of game code without modifying it.

7 Conclusions and Future Work

We described a general and scalable audit framework for massively multi-player 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 player. The main advantage of our random checking algorithm is that the audit server only needs to repeat a small number of game renderings to catch cheaters with high probability. This randomization significantly saves the computation costs on the audit server side.

An emerging architecture for multi-player online games is based on peer-to-peer overlay networks where there is no central game server. Instead, a player communicates directly with other players to coordinate their interactions and states among themselves [2, 12]. Security enforcement via auditing has not been studied in this type of settings. In particular, it is interesting to explore whether or not it is feasible to design a robust and distributed audit protocol that allows peers to audit each other. Such an audit protocol should be able to secure against colluding peers that do not report cheating behaviors of each other.

References

1. N. E. Baughman and B. N. Levine. Cheat-proof playout for centralized and distributed online games. In *IEEE INFOCOM*, pages 104–113, 2001.

2. N. E. Baughman, M. Liberatore, and B. N. Levine. Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Trans. Netw.*, 15(1):1–13, 2007.
3. B. D. Chen and M. Maheswaran. A cheat controlled protocol for centralized online multiplayer games. In *Proceedings of the 3rd Workshop on Network and System Support for Games (NETGAMES)*, pages 139–143. ACM, 2004.
4. B. D. Chen and M. Maheswaran. A fair synchronization protocol with cheat proofing for decentralized online multiplayer games. In *3rd IEEE International Symposium on Network Computing and Applications (NCA)*, pages 372–375, 2004.
5. S. B. Davis. Why cheating matters. In *Game Developer's Conference*, 2003. `http://www.secureplay.com/papers/docs/WhyCheatingMatters.pdf`.
6. W. Du, J. Jia, M. Mangal, and M. Murugesan. Uncheatable grid computing. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 4–11.
7. S. Haber, Y. Hatano, Y. Honda, W. Horne, K. Miyazaki, T. Sander, S. Tezoku, and D. Yao. Efficient signature schemes supporting redaction, pseudonymization, and data deidentification. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 353–362, 2008.
8. P. Harding-Rolls. Western world mmog market: 2006 review and forecasts to 2011, March 2007. Management Report. Screen Digest.
9. S. Jha, S. Katzenbeisser, C. Schallhart, H. Veith, and S. Chenney. Enforcing semantic integrity on untrusted clients in networked virtual environments. In *IEEE Symposium on Security and Privacy*, pages 179–186, 2007.
10. C. Joslin, T. D. Giacomo, and N. Magnenat-Thalmann. Collaborative virtual environments: From birth to standardization. *IEEE Communications Magzine*, pages 28–33, April 2004.
11. C. Joslin, I. S. Pandzic, and N. Magnenat-Thalmann. Trends in networked collaborative virtual environments. *Computer Communications*, 26(5):430–437, 2003.
12. B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE INFOCOM*, 2004.
13. R. C. Merkle. Protocols for public key cryptosystems. In *Proceedings of the 1980 Symposium on Security and Privacy*, pages 122–133. IEEE Computer Society Press, 1980.
14. R. C. Merkle. A certified digital signature. In G. Brassard, editor, *Proc. CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer-Verlag, 1989.
15. S. Pearson. Trusted computing: Strengths, weaknesses and further opportunities. In *Proceedings of the Third International Conference on Trust Management for Enhancing Privacy (iTrust)*, volume 3477 of *Lecture Notes in Computer Science*, pages 305–320, 2005.
16. M. Pritchard. 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. X. Wang, H. Yu, and Y. L. Yin. Efficient collision search attacks on sha-0. In *Advances in Cryptology – CRYPTO*, pages 1–16, 2005.
19. World of Warcraft. `http://www.worldofwarcraft.com/index.xml`.
20. J. Yan. Security design in online games. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, page 286, Washington, DC, USA, 2003. IEEE Computer Society.