Enforcing Semantic Integrity on Untrusted Clients in Networked Virtual Environments (Extended Abstract) *

Somesh Jha[1] Stefan Katzenbeisser[2] Christian Schallhart[2] Helmut Veith[2] Stephen Chenney[3]
[1] University of Wisconsin [2] Technische Universität München [3] Emergent Game Technology

Abstract

*In the computer gaming industry, large-scale simulations of realistic physical environments over the Internet have attained increasing importance. Networked virtual environments (*NVEs*) are typically based on a client-server architecture where part of the simulation workload is delegated to the clients. This architecture renders the simulation vulnerable to attacks against the semantic integrity of the simulation: malicious clients may attempt to compromise the physical and logical rules governing the simulation, or to alter the causality of events. This paper initiates the systematic study of semantic integrity in* NVEs *from a security point of view. We present a new provably secure semantic integrity protocol which enables the server system to audit the local computations of the clients on demand.*

1. Introduction

Networked Virtual Environments (NVEs) are software systems in which users feel immersed in an artificial world, typically viewed through a 3D rendering. The most widely deployed examples of NVEs are networked interactive games, such as *Unreal Tournament* [9] as well as social NVEs such as *Second Life* [12]. Some estimates [5] claim the real-world value of online game assets exceeds $2 billion, while the daily real-world transactions between Second Life users are reported at $500,000 [13].

Because of the high immersion in the game, NVE players are very sensitive to perceived cheating, and cheating in online games is therefore one of the fastest ways to destroy a community and commercial reputation. Cheating is an attack on the *semantic integrity* of the online community: a malicious user may attempt to compromise the logical rules governing the simulation, or to alter the causality of events a posteriori. In this paper, we initiate the systematic study of security issues in NVEs and present security protocols which prevent malicious participants from compromising the semantic integrity of the NVE.

*Supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

NVE System Architecture. We consider *remote access* NVE*s based on the client-server model* [11, 18] where the authoritative and central version of the NVE state is maintained by the server system StateServer. The clients, $\mathsf{cli}_i, 1 \leq i \leq n$, connect to the server over the Internet, and receive state updates to maintain their local models of the environment. Seeing a graphical representation of the model on cli_i, the user can initiate an action (e.g., pick up an object, send a text message, etc.) which cli_i communicates to the server in the form of a *state update request*. StateServer checks whether the requested actions are compliant with the rules of the NVE, and sends an authoritative state update message to cli_i. This update message contains the requested state update of cli_i as well as all other changes to the central state that occurred since the last update message was sent to cli_i. Finally, cli_i updates its local state according to the answer received from the server system. The procedure starting with the update request and ending with the local state update is called the client cycle.

There is no practical measure to prevent that client software is modified by malicious participants. Simple modifications include exposing supposedly hidden state or modifying damage done by weapons. Consequently, any security assessment of NVEs must assume that all clients are untrusted. Thus, NVEs with remote access not only have to cope with the deficits of the networking infrastructure (long transmission times and frequent packet loss [17]), but also with *malicious clients attacking the* NVE .

The Semantic Gap and Security. In NVEs with thousands of players, limited internet bandwidth and limited server capabilities make it necessary to off-load computational work to the clients. Clearly, it is most beneficial to transfer the most computationally demanding tasks, in particular rendering of 3D images and simulations of natural phenomena. Consequently, the client computations must be trusted to embody some of the rules of the simulated world. For example, a physically-based simulation of a user's vehicle would be done on the client, and only this simulation

2007 IEEE Symposium on Security and Privacy(SP'07)
0-7695-2848-1/07 $20.00 © 2007 IEEE

can tell us if the vehicle stays on the road as it rounds a bend. The server has only an *abstract* representation of the world: the user is in a vehicle at a certain location moving at a certain speed. Only the client is computing the *concrete* outcome of the simulation step. We refer to the difference between server and client knowledge as the *semantic gap*.

The semantic gap is the primary means by which malicious clients subvert semantic integrity. Exploiting the semantic gap, they can submit *spurious updates* that are consistent with the NVE on the abstract level, but violate the NVE semantics at the concrete level. In our vehicle example, the client may pretend that the vehicle rounds the bend, even though the client simulation indicates that it crashes.

Closing all semantic gaps requires a very extreme form of NVE, in which final rendered images are computed on the server and securely sent to clients. This is totally impractical – it takes all of the resources of a dedicated graphics card to compute one image on a client, while a server would have to compute thousands of these images, not to mention the bandwidth. On the contrary, economic concerns demand very aggressive movement of simulation from the server to the client. Given that we cannot close the semantic gap, our goal is to detect the presence of spurious updates. This is challenging because the trustworthy server does not have the clients' complete local states, including the rendered images, and has no hope of obtaining all such states at every time step.

Technical Contribution. The main technical contribution of this paper is a set of provably secure protocols that maintain the semantic integrity of the NVE, even in the presence of maliciously modified clients. Our approach is based on an efficient *audit procedure* that is performed repeatedly and randomly on the NVE clients. During the audit process, it is verified whether the *concrete* state updates performed by the client in a specific time frame are valid according to the NVE semantics.

As our solution was designed from an engineering perspective, it has several favorable practical properties: The protocol incurs very low additional network traffic, and uses reliable and time critical network transmissions only for a few small messages. Finally, the protocol can be integrated with existing middleware approaches quite easily.

Related Work on Security. Audit trails were applied in e-commerce (e.g. [14]). Bellare and Yee [3] identified *forward security* as the key security property for audit trails, i.e., even if an attacker completely compromises the auditing system, the attacker should not be able to forge audit information referring to the past. The protocols described in this paper follow the principles of audit trails, but account for the specific particularities of NVE environments.

The approach taken is fundamentally optimistic: we allow cheating to happen, but aim at later detection. Under the assumption that cheating does not occur too often, this approach incurrs only low detection overhead. The approach is thus related to optimistic fault tolerance [20]. Replication techniques for Byzantine fault tolerance [4, 16] also seem applicable to our problem. However, since the client has complete control of the replicas, these techniques cannot address the semantic-integrity problem.

Few papers have studied security issues in online games, see e.g. [2, 6, 8, 21, 22]. While Pritchard [15] deals with semantic attacks, his approach requires each client to run the entire simulation, which does not scale for MMOGs.

2. Threat Analysis

In this paper, we are concerned with *semantic subversion* of NVEs, and do not consider *system security attacks* [1, 19] or *meta-strategies* such as collusive collaboration or mobbing. A semantic attack is an attack targeted at circumventing or subverting the rules (i.e., the semantics) of the NVE. We classify semantic attacks as follows:

1. **Semantic Integrity Violation.** The attacks violates the physical and logical laws of the NVE. All attacks in this class involve maliciously modified software:
   (a) **Rule Corruption:** The malicious client attempts to modify the simulation in a way that is illegal but plausible to the server system. For example, the client modifies their vehicle physics system to allow higher speeds without negative road-holding consequences. The server is not running the complex vehicle simulation, so it does not know precisely what the vehicle should be doing.
   (b) **Causality Alteration:** The malicious client attempts to withdraw previous state changes to obtain unfair advantages, i.e., the client attempts to "rewrite its history". For example, position information could be changed to avoid taking damage from an explosion, after the explosion had happened and damage was determined by the client.
2. **Client Amplification:** The client runs modified software to exploit the possibilities of the NVE in an unintended manner. We distinguish two categories:
   (a) **Sniffing:** The malicious client exposes information which has to be downloaded for technical reasons but is not intended to be observable immediately. For example, a client can be modified to render opaque walls as transparent.
   (b) **Agents:** The malicious client enhances the intended capabilities of the human participant. For example, an agent can employ search strategies

2007 IEEE Symposium on Security and Privacy(SP'07)
0-7695-2848-1/07 $20.00 © 2007 IEEE

to guide the player, or log and replay successful prior actions.

We consider Semantic Integrity Violation the most important NVE-specific class of attacks which needs to be treated at the protocol level. The protocols presented in this paper consider both rule corruption and causality alteration attacks. To do so, the protocols enforce the following two requirements on the client behavior:

- **Rule Compliance.** The client has to follow the semantical rules of the NVE. This prevents rule corruption.
- **Monotone History.** The actions of the client must be irrevocable and undeniable. This condition prevents causality alteration.

3. Unsecured Client Cycle

In this section, we review the state update mechanism that is commonly implemented in NVEs that maintain a central abstract state ASTATE. Depending on the spatial position of cli_i, only a portion $\text{ASTATE}[\mathsf{cli}_i]$ of the entire state is relevant for the cli_i. The relevant portion of the abstracted and centrally maintained state is transfered to the client. Locally, this abstract state is expanded to a concrete state at the client.

Given an abstract state s, $\gamma(s)$ denotes the set of possible concretizations. If S is a concrete state, then $\alpha(S)$ is the corresponding unique abstract state. The pair $\alpha()/\gamma()$ can be naturally viewed as a Galois connection between the set of abstract and concrete states [7], i.e., $S \in \gamma(\alpha(S))$ and $s = \alpha(S)$ for any $S \in \gamma(s)$. When connecting to the NVE, cli_i receives a concrete state $S \in \gamma(\text{ASTATE}[\mathsf{cli}_i])$ to initialize its local state $\text{STATE}[\mathsf{cli}_i]$. Afterwards, cli_i maintains $\text{STATE}[\mathsf{cli}_i]$ locally and only receives abstract updates.

If cli_i wishes to change its state, it informs the StateServer in order to update ASTATE. For this purpose, cli_i computes a compact state update description Δ between the current state $\text{STATE}[\mathsf{cli}_i]_t$ and the intended next state; we call Δ a *diff*. $S' = S + \Delta$ denotes the application of a diff Δ on state S, yielding state S'.

We apply $\alpha()$ and $\gamma()$ not only to states, but also to diffs. In particular, $\alpha(\Delta)$ denotes the abstraction of a diff Δ. For $S' = S + \Delta$ holds, we also require $\alpha(S') = \alpha(S) + \alpha(\Delta)$. Not every concretization Δ of an abstract diff δ is applicable to a given concrete state S. Therefore, $\gamma(S, \delta)$ denotes the set of concretizations of an abstract diff δ which can be applied to S. More precisely, if $S' = S + \Delta$, then $\Delta \in \gamma(S, \alpha(\Delta))$ and for all $\Delta' \in \gamma(S, \alpha(\Delta))$, we get $\alpha(S + \Delta') = \alpha(S')$.

One client cycle consists of the following steps: The cli_i sends an abstraction $\delta = \alpha(\Delta)$ of the concrete changes Δ to StateServer. This abstract diff δ contains the changes requested by cli_i and is called *request diff*. Then StateServer checks which changes in δ are valid according to the semantics of the NVE and assembles a reply δ' which authorizes all valid request of cli_i and contains all updates performed by other clients present in the NVE. Upon receipt of δ', cli_i computes a concretization $\Delta' \in \gamma(\text{STATE}[\mathsf{cli}_i], \delta')$ and updates its own state by computing $\text{STATE}[\mathsf{cli}_i]_{t+1} = \text{STATE}[\mathsf{cli}_i]_t + \Delta'$. The response δ' of the StateServer is called *authoritative diff*.

If the clients behave according to the NVE specification, this protocol suffices to consistently maintain the states of the clients and the server. However, in case of malicious clients, the protocol is susceptable to a *semantic integrity* violation, as StateServer is only able to check whether the *abstract* state updates $\delta = \alpha(\Delta)$ are consistent with its *abstract* state.

4. Secure Semantic Integrity Protocol (SSIP)

In this section, we show how to amend the basic client cycle described above with cryptographic mechanisms to prevent semantic integrity violation attacks. Our approach uses an audit procedure, which is performed by a dedicated and fully trusted AuditServer. During each client cycle, the client reliably sends a piece of evidence (containing a hash of the applied *concrete* state update) as action commitment to AuditServer. From time to time, the cli commits to a concrete state; these states will serve as possible starting states for the audit process.

Note that our security model assumes that AuditServer is fully trusted. In particular, the protocols do not provide non-repudiation: a cheating audit server could frame innocent clients by wrongly claiming that they behaved badly. However, we do not consider this case, as we believe that it is not a practical situation in commercial NVEs.

When auditing is initiated, AuditServer asks a cli to provide a sequence of concrete state updates for a specific time frame together with an initial concrete full state. Based on this information, AuditServer simulates the requested segment of the cli computation and checks both its compliance to the NVE rules and its consistency with the action commitments sent previously.

Audit Cycles. The auditing process is subdivided into *audit cycles,* where each audit cycle consists of exactly l client cycles. At each l-th client cycle a new audit cycle is started. In this paper, we assume for simplicity that l is a system-wide announced and agreed on parameter.

At the beginning of each audit cycle, the client sends a hash of the concrete full state as action commitment to AuditServer. As this hash may be costly to compute because of the large state description, this message has to arrive only within the current audit cycle (i.e., within the next l client cycles). During each client cycle, the client sends

additionally an action commitment of the applied concrete diff; as the diff is usually small, we require that this message arrives at AuditServer during the same client cycle.

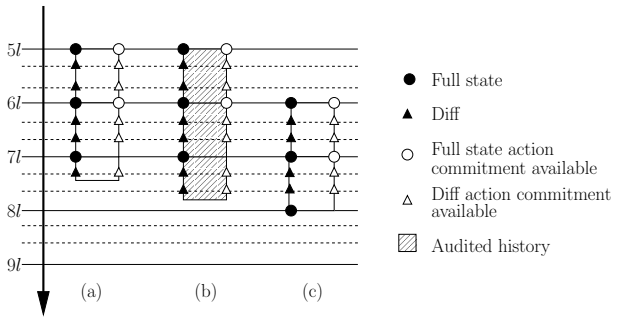

Figure 1. Audited History – "Sliding Window"

While StateServer only keeps the current abstracted central state, the clients maintain their current concrete state *and retain a history of previous states in a local buffer,* containing up to 3 full states and $3l$ diffs. In particular, the cli has to retain a copy of the complete state at the beginning of each new audit cycle together with diffs between the states of intermediate client cycles. All buffer content older than three audit cycles on the client side can be safely deleted. The buffer thus describes a "sliding window" which contains the state history of the last $2l + 1$ to $3l$ client cycles, i.e., the last two full audit cycles and the current one. The *sliding window* which is maintained at client cycle $t_0 \geq 2l$ contains the states $S_{t_a}, S_{t_a+l}, S_{t_0}$ as well as all the intermediate diffs $\Delta'_{t_a+1}, \ldots, \Delta'_{t_0}$ where $t_a = \left\lfloor \frac{t_0}{l} - 2 \right\rfloor l$. Thus, t_a denotes the expiration time for client side audit information. The client also stores all messages received from the server within the time interval determined by the sliding window. Figure 1 illustrates the gradual change of the buffer for one client. The symbol • represents a fully saved state, whereas ▲ represents a concrete diff, both saved at the client. On the other hand, ∘ and △ represent action commitments of full states and diffs which are available at the AuditServer.

Audit Process. During the auditing, cli must prove that its actions during the last two finished audit cycles and the current audit cycle are compliant to the rules of the NVE. For this purpose, cli sends the state information of the current sliding window together with all corresponding StateServer messages to AuditServer. Now, AuditServer checks whether the received state information matches the previously submitted action commitments, whether the client computation is compliant to the rules of the NVE and whether the client correctly committed itself to the starting states of all audit cycles contained in the audited period. The audit results in a positive verdict if and only if all checks succeed. Note that the third condition is of central importance, as this prohibits the client from cheating on future audit starting states.

Crucial to the correctness of the audit process is the enforcement of the timing conditions for the action commitments. The action commitment of a diff *must* arrive reliably within the current client cycle, whereas action commitments of full states must only arrive when the current audit cycle is completed. In Figure 1 the action commitments (represented by △) for diffs are available at the AuditServer immediately. In contrast, the action commitment ∘ for the full state $7l$ becomes available when the system enters state $8l$.

Note that the late availability of the full state action commitment messages requires the audit process to audit at least two full audit cycles, as otherwise the semantic integrity of the future audit starting points cannot be checked.

Protocol Description. Secure integrity enforcement is performed by three protocols ***Initialize***, ***StatusUpdate*** and ***Audit***. The protocol ***Initialize*** is performed whenever a client joins the NVE, whereas ***StatusUpdate*** is executed at each client cycle. Finally, ***Audit*** implements the auditing mechanism.

For the sake of simplicity, we present the protocol for a single client cli that interacts with StateServer and AuditServer. For multiple clients, the protocol is processed asynchronously in parallel. Sending a message unreliably will be denoted by $\rightsquigarrow$. Sending a message reliably that must arrive before the next t-th client cycle is initiated, will be denoted by $\hookrightarrow_t$. Unreliable messages may be dropped or delivered with delay. However, we assume that no packet corruption occurs.

In the protocols we use a Message Authentication Code (MAC) and a collision-free hash function as cryptographic primitives. For computing MAC-tags, an appropriate key $k = \text{GenMac}(1^n)$ is generated where n is the security parameter. Then, a tag t for a message m is computed with $t = \text{SignMac}(k, m)$, whereas the verification algorithm is written as $\text{VerifyMac}(k, m, t) \in \{\text{true}, \text{false}\}$. We write $M = \text{AuthMsg}(k, m, \text{cli})$ as an abbreviation for $m \,\|\, \text{SignMac}(k, m \,\|\, \text{cli})$, where $\|$ denotes string concatenation. Furthermore, we will denote with $M^{(1)}$ and $M^{(2)}$ the two parts of the message M, i.e., $M^{(1)} = m$ and $M^{(2)} = \text{SignMac}(k, m \,\|\, \text{cli})$. The hash function $\text{CFHash}_h(m)$ is chosen from a collection of collision-free hash functions. Let $h = \text{GenCFHash}(1^n)$ be its index, where n is the security parameter. For the sake of simplicity we will abbreviate $\text{STATE}[\text{cli}]_t$ with S_t. The protocols use a single MAC key k which is mutually agreed between the state server and the audit server and is used to authenticate status updates sent from StateServer to cli.

1. cli initializes $t := 0$ and sends an initialization request to StateServer.
2. StateServer $\rightsquigarrow$ AuditServer : $k := \text{GenMac}(1^n)$
3. AuditServer $\rightsquigarrow$ cli : $h := \text{GenCFHash}(1^n)$
4. StateServer chooses $S \in \gamma(\text{ASTATE}[\text{cli}])$
5. StateServer $\rightsquigarrow$ cli : $M_0 := \text{AuthMsg}(k, S \,\|\, n_0, \text{cli})$
6. cli sets $S_0 := S$
7. cli $\hookrightarrow_l$ AuditServer : $Q_0 := \text{CFHash}_h(S_0)$

Figure 2. Protocol *Initialize*

Protocol *Initialize*. This protocol initializes the state of a cli joining the NVE (see Figure 2). Upon opening a connection to StateServer, an appropriate MAC-key k as well as an index h for the collision-free hash function are generated and distributed. Then, the client receives the relevant status information together with a randomly generated nonce n_0 and a MAC of the message. At this point the state server transmits a *concrete* state $S \in \gamma(\text{ASTATE}[\text{cli}])$ to the client. The client initializes its local state S_0 with S. This is the only point, besides the audit procedure, where a concrete state is transmitted. Finally, the client sends as evidence a hash of its state S_0 reliably to the audit server; as the hash of the concrete state may be costly to compute, this hash must only arrive before the lth client cycle is initiated.

Protocol *StatusUpdate*. After initialization, the client uses this protocol to update its local state in each client cycle (see Figure 3). Suppose the client is in state S_t and wants to change its state according to the diff Δ_{t+1}. To initiate the update protocol, the client reliably sends an abstracted request diff $\delta_{t+1} = \alpha(\Delta_{t+1})$ to StateServer. The server checks whether this request conforms to its the current ASTATE and computes a new authoritative diff δ'_{t+1}. This diff δ'_{t+1} contains the legitimate changes of δ_{t+1} and changes caused by other clients. StateServer updates its centrally managed state ASTATE according to δ'_{t+1} and returns $M_{t+1} := \text{AuthMsg}(k, \delta'_{t+1} \,\|\, n_t + 1, \text{cli})$ (consisting of the diff, an incremented nonce, and a MAC) to the client.

The client now computes a concrete update $\Delta'_{t+1} \in \gamma(S_t, \delta'_{t+1})$ and applies it to S_t to enter the next state $S_{t+1} = S_t + \Delta'_{t+1}$. Finally the client sends a hash $D_{t+1} := \text{CFHash}_h(\Delta'_{t+1})$ as action commitment reliably to the AuditServer before the next client cycle is started. At the beginning of each audit cycle, the client sends a hash $Q_t := \text{CFHash}_h(S_t)$ of its full state to AuditServer. This message is sent reliably but must only arrive within the current audit cycle, i.e., within the next l client cycles.

1. cli computes a desired status change Δ_{t+1} and its abstraction $\delta_{t+1} = \alpha(\Delta_{t+1})$
2. cli $\rightsquigarrow$ StateServer : δ_{t+1}
3. Upon receiving δ_{t+1}, StateServer computes a new δ'_{t+1} and updates its ASTATE accordingly
4. StateServer $\rightsquigarrow$ cli : $M_{t+1} := \text{AuthMsg}(k, \delta'_{t+1} \,\|\, n_t + 1, \text{cli})$
5. cli chooses and stores $\Delta'_{t+1} \in \gamma(S_t, \delta'_{t+1})$ and computes $S_{t+1} = S_t + \Delta'_{t+1}$
6. cli $\hookrightarrow_1$ AuditServer : $D_{t+1} := \text{CFHash}_h(\Delta'_{t+1})$
7. cli increments t
8. if $t \bmod l = 0$
   (a) cli deletes all Δ'_{t-i} with $2l \le i < 3l$ as well as the full state S_{t-3l} (if $t \ge 3l$).
   (b) cli stores S_t and starts to compute $Q_t := \text{CFHash}_h(S_t)$.
   (c) After computation of Q_t, cli $\hookrightarrow_l$ AuditServer : Q_t.

Figure 3. Protocol *StatusUpdate*

Protocol *Audit*. During the audit protocol, AuditServer validates the computation of one cli (see Figure 4). In particular, AuditServer checks whether the client can present concrete state updates that match the action commitments received so far and are consistent with the NVE rules. The auditing protocol starts with an audit message sent to the cli during client cycle t_0. The client first computes the starting point t_a of the audit. The client then (unreliably) sends the concrete state S_{t_a} as well as all diffs Δ'_i and messages M_i for $t_a + 1 \le i \le t_0$ to the AuditServer. Then, AuditServer checks, using the action commitment messages D_i and Q_i submitted by the client before, whether the client adhered to the NVE semantics: AuditServer checks whether all Δ'_i are suitable concretizations of δ'_i sent by the state server in message M_i, whether all state server messages M_i ($t_a + 1 \le i \le t_0$) are unmodified and whether all action commitment messages (D_t and Q_t) submitted by the client beforehand are valid. To perform the latter operation, the AuditServer checks the hashes in the messages D_i, $t_a + 1 \le i \le t_0$, and the hashes of the full states S_{t_a} and S_{t_a+l}, contained in the messages Q_{t_a} and Q_{t_a+l} (by the timing conditions, these messages are already available to AuditServer). If the first audit cycle is audited ($t_a = 0$), then cli is required to present $M_0^{(2)} = \text{SignMac}(k, S_0 \| \text{cli})$ to AuditServer additionally to prove that the initial state S_0 has been authorized by the StateServer. If all checks pass, the client is considered honest.

1. AuditServer $\rightsquigarrow$ cli : $audit \,\|\, t_0$
2. cli computes $t_a = \left\lfloor \frac{t_0}{l} - 2 \right\rfloor l$
3. cli $\rightsquigarrow$ AuditServer :
   $S_{t_a} \,\|\, \Delta'_{t_a+1} \,\|\, \ldots \,\|\, \Delta'_{t_0} \,\|\, M_{t_a+1} \,\|\, \ldots \,\|\, M_{t_0}$
4. AuditServer computes $\hat{S}_{i+1} = \hat{S}_i + \Delta'_{i+1}$ for $i = t_a, \ldots, t_0 - 1$ where $\hat{S}_{t_a} = S_{t_a}$
5. For all $i = t_a+1, \ldots, t_0$, AuditServer checks whether Δ'_i is chosen from $\gamma(\hat{S}_i, \delta'_i)$ compliant with the rules of the NVE, where δ'_i is taken from the message M_i
6. For all $i = t_a+1, \ldots, t_0$, AuditServer checks whether
   (a) VerifyMac$(k, M_i^{(1)} \,\|\, \text{cli}, M_i^{(2)}) =$ true and
   (b) CFHash$_h(\Delta'_i) = D_i$
7. AuditServer checks whether CFHash$_h(S_{t_a}) = Q_{t_a}$ and CFHash$_h(\hat{S}_{t_a+l}) = Q_{t_a+l}$.
   If $t_a = 0$, cli $\rightsquigarrow$ AuditServer : $M_0^{(2)}$ and AuditServer checks VerifyMac$(k, S_0 \| \text{cli}, M_0^{(2)}) =$ true.
8. AuditServer accepts the computations of cli if and only if all tests in steps 5 to 7 passed.

Figure 4. Protocol *Audit*

Security. It can be shown that the audit protocol enforces honest client behavior if MAC tags are unforgeable and the employed hash function is collision-resistant. In particular, the protocol assures rule compliance and monotone history of the clients (as introduced in Section 2) within audited time periods. We omit a formal proof for space reasons and refer the interested reader to [10].

5. Conclusion and Future Work

In this paper, we have argued that networked virtual environments are an emerging network technology which has not been subject to rigorous security investigations. We have identified *semantic integrity* as a one central security problem in NVEs. Untrusted and malicious clients may utilize the fact that the central NVE server can—due to the limited computing power and the disruptions in the network connection—only maintain an abstracted version of the NVE state. To overcome this problem, we have introduced a new *provably secure* audit trail mechanism which is able to verify the compliance of the client computation. Although we allow autonomous clients, our protocols assure that regularly cheating clients will be identified with a high probability. The audit mechanism proposed in this paper can be seamlessly integrated into current NVE architectures and incurs little engineering and resource overhead.

References

[1] R. Anderson. *Security Engineering*. Wiley, 2001.

[2] N. Baughman and B. Levine. Cheat-Proof Playout for Centralized and Distributed Online Games. In *Proc. 20th IEEE INFOCOM*, pages 104–113, 2001.

[3] M. Bellare and B. Yee. Forward Integrity for Secure Audit Logs. Technical report, UCSD, 1997.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[5] E. Castronova. *The Business and Culture of Online Games*. University of Chicago Press, 2005.

[6] B. Chen and M. Maheswaran. A fair synchronization protocol with cheat proofing for decentralized online multiplayer games. In *Proc. 3rd IEEE NCA*, pages 372–375, 2004.

[7] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4th POPL*, pages 238–252, 1977.

[8] S. Davis. Why Cheating Matters. Cheating, Game Security and the Future of On-line Gaming Business. In *Game Developers Conference*, 2003.

[9] E. Games. Unreal Tournament. `http://www.unrealtournament.com`, 1999.

[10] S. Jha, S. Katzenbeisser, C. Schallhart, H. Veith, and S. Chenney. Enforcing semantic integrity on untrusted clients in networked virtual environments. Cryptology ePrint Archive, Report 2007/056, 2007. `http://eprint.iacr.org/2007/056`.

[11] C. Joslin, T. D. Giacomo, and N. Magnenat-Thalmann. Collaborative Virtual Environments: Form Birth to Standardization. *IEEE Communications*, pages 28–33, April 2004.

[12] L. Lab. Second Life. `http://secondlife.com`, 2003.

[13] A. Pasick. US Congress launches probe into virtual economies. Reuters, October 15 2006.

[14] J. Peha. Electronic Commerce with Verifiable Audit Trails. In *Proceedings of INET'99, Internet Society*, 1999.

[15] M. Pritchard. How to Hurt the Hackers: The Scoop on the Internet Cheating and How You Can Combat It. *Game Developer Magazine*, June 2000.

[16] M. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the ACM Conference on Computer and Communications Security*, 1994.

[17] S. Singhal. *Effective Remote Modelling in Large-Scale Distributed Simulation and Visualization Environments*. PhD thesis, Stanford University, 1996.

[18] S. Singhal and M. Zyda. *Networked Virtual Environments: Design and Implementation*. Addison-Wesley, 1999.

[19] W. Stallings. *Cryptography and Network Security*. Prentice Hall, 2003.

[20] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[21] J. Yan. Security Issues in Online Games. *The Electronic Library: international journal for the application of technology in information environments*, 20(2), 2002.

[22] J. Yan and H. Choi. Security Design in Online Games. In *Annual Computer Security Applications Conference*, 2003.