Enforcing Semantic Integrity on Untrusted Clients in Networked Virtual Environments [*]

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

*During the last years, large-scale simulations of realistic physical environments which support the interaction of multiple participants over the Internet have become increasingly available and economically viable, most notably in the computer gaming industry. Such systems, commonly called networked virtual environments (*NVES*), are usually based on a client-server architecture where for performance reasons and bandwidth restrictions, the simulation is partially delegated to the clients. This inevitable architectural choice 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 a posteriori.*

In this paper, we initiate the systematic study of semantic integrity in NVES *from a security point of view. We argue that naive policies to enforce semantic integrity involve intolerable network load, and are therefore not practically feasible. We present a new* provably secure *semantic integrity protocol based on cryptographic primitives which enables the server system to audit the local computations of the clients on demand. Our approach facilitates low network and* CPU *load, incurs reasonable engineering overhead, and maximally decouples the auditing process from the soft real time constraints of the simulation.*

1 Introduction

Networked Virtual Environments (NVEs) are software systems in which users feel immersed in an artificial world, typically viewed through a three-dimensional graphical rendering. The most widely deployed examples of NVEs are networked interactive games, such as *Unreal Tournament* [1], in which remotely-located players compete against each other using standard Internet connections. Social NVEs, such as *Second Life* [2], have recently emerged and are drawing corporate marketing resources to their online community [3].

Security in the form of cheat prevention - ensuring that everyone is playing by the same rules - is among the primary technical and business imperatives for NVE operators. Beyond ensuring the reliability and integrity of their systems, operators must maintain a community of repeat participants who generate a steady revenue stream. Players are typically very sensitive to perceived cheating, and enabling cheating in online games is one of the fastest ways to destroy a community and commercial reputation.

Massively multi-player online games (MMOGs), in which the number of concurrent participants may be measured in the hundreds of thousands, are seen by many as the greatest source of growth in online gaming. Examples include *Eve Online* [4] and *World of Warcraft* [5], the latter with over 6 million subscribers. The

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

key technical feature of these worlds is persistence, and this has spawned entire virtual economies tied to the real world economy through the sale of game assets. Some estimates [6] 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 [7]. In-game cheating threatens the value of game assets if expensive goods are made common through cheating, and hence has real-world economic implications.

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. Such attacks have been repeatedly reported in the trade publications for gaming applications, yet are rarely found in the academic literature. The goal of the current paper is to *initiate the systematic study of security issues in* NVE*s* and to *present security protocols which prevent malicious participants from compromising the semantic integrity of the NVE.*

Unlike online applications in which clients lie and cheat about their real world characteristics, in an NVE the application software completely defines the rules of the environment: virtual characters are created in the space and live their entire virtual lives according to the software implementation of the environment. Secure software systems provide the opportunity to eliminate a broad range of cheats, because, in theory, any character's deviation from the pre-programmed rules of the game can be detected. Unfortunately, practical limits on the design and implementation of NVEs make it difficult to guarantee their semantic integrity.

NVE System Architecture. In this paper we are only concerned with *remote access* NVE*s based on the client-server model* [8, 9]. This includes almost all widely-used NVEs. In a client-server NVE architecture, the authoritative and central version of the state of the NVE is maintained by the server system $\mathsf{StateServer}$. The clients, $\mathsf{Client}_1, \ldots, \mathsf{Client}_n$, connect to the server over a wide-area, insecure network, and receive state updates that are used to reconstruct their local model of the environment. The user is presented with a graphical representation of the model and can provide input in response to what they see (pick up an object, send a text message, etc.). The resulting events, in the form of state update requests, are sent back to the server. We refer to the processing of state updates and the evolution of the world model as *simulation*.

Each time Client_i wants to update the shared state of the NVE, it has to send a corresponding request to $\mathsf{StateServer}$. $\mathsf{StateServer}$ checks whether the requested actions are compliant to the rules of the NVE. If this is the case, $\mathsf{StateServer}$ sends to Client_i an authoritative state update message that contains (as acknowledgment) the requested state update of Client_i as well as all changes to the central state that occurred since the last update message was sent to Client_i. Finally, Client_i updates its local state according to the answer received from the server system.

We define a *client cycle* as the procedure which starts with the computation of the update request by Client_i and ends with the client state update according to the server response. In a typical NVE application the client cycle is repeated at around 10Hz. Different portions of the state may be updated at different rates to balance interactivity versus bandwidth. Interpolation is used to present the user with a higher visible update rate.

Client software can be 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 [10]), but also with *malicious clients attacking the* NVE . These attacks take many forms, which we describe in Section 2.

The Semantic Gap and Security. One of the most important business metrics for an NVE is the number of clients per server. Each client represents monthly subscription revenue, while each server represents a cost, and providers aim to maximize their net revenue per client. Each user represents a computational load, so servers must be added as clients are added, or the work per client must be reduced.

Tightly interconnected server clusters are used to add servers while maintaining the appearance of a single StateServer machine. Since all resources within the cluster are solely dedicated to the NVE, we will assume that they are mutually trusted. Consequently, clustering does not directly affect the security of the NVE. Clustering enables growth in client numbers, but it does not improve the client per server ratio. Clustering also fails to provide for NVEs that enable users to run their own servers, such as *Unreal Tournament*.

The practical way to reduce server workloads is to off-load more work to the client. Clearly, it is most beneficial to transfer the most computationally demanding tasks: rendering of 3D images and simulations of natural phenomenon are two examples of tasks that are almost always computed on the client. Users actively prefer this approach because it allows them to improve their personal experience by investing in their own computational resources.

Computing on the client is only of benefit if the server does not repeat the work, but then the client computations must be trusted to embody some of the rules of the world. For example, a physically-based simulation of a user's vehicle would be done on the client, and only this simulation 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. Such a client can successfully submit *spurious updates* that are consistent with the rules of the NVE on the abstract level, but violate the NVE semantics at the concrete level. In our vehicle example, the client could inform the server 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, we aim instead 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 state at every time step.

Engineering Requirements. The computational workloads of NVEs not only lead to security problems via the semantic gap, they also place requirements on the design of any security solution:

- **Minimal and Scalable Resource Overhead:** The overhead caused by the security solution should be as small as possible and scale well with respect to the number of participants and the level of security to be enforced. While this sounds like an obvious requirement, we want to stress that the typical MMOG client application is heavily optimized to exploit all resources in the drive for a compelling experience. *Therefore, any approach which causes significant CPU or network overhead at the client- or server-side is impractical.*
- **Minimal Quality of Service Requirements.** A security solution should utilize unreliable network services for as many messages as possible. *Only a few important messages should require timely and reliable send operations.*
- **Minimal Engineering Overhead:** NVEs are complex software systems which usually consist of multiple large software packages, some of which may be third-party middleware. Thus, it is often infeasible to modify an existing NVE to meet strict security requirements. *A security solution should be as independent as possible from the simulation and should only affect small portions of the code.*

Technical Contribution. The main technical contribution of this paper is a set of provably secure protocols that maintain the states of the clients and the server consistently and securely, even in the presence of maliciously modified clients. The 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. The solution meets all above stated requirements:

- The protocols enforce semantic integrity on the NVE clients, while allowing a central abstracted state and autonomous clients. We prove in appendix A that the protocols are secure under standard cryptographic assumptions.
- The solution incurs very low additional network traffic, and requires the transmission of complete client states over the network *only during the audit process*. More precisely, our solution requires only a few additional bytes per client cycle which is a negligible quantity in comparison to the other messages in the client cycle. The security overhead consists solely of a hash which is considered secure at a size of about 20 bytes. On the other hand, a modern MMOG requires roughly one kilobyte per client and cycle. Note that with thousands of clients, bandwidth is a bottleneck mainly at the server side.
- Our solution uses reliable and time critical network transmissions only for a few small messages. All other messages, in particular the complete audit process, can be implemented solely using unreliable send operations.
- The audit process is completely independent of the (time critical) simulation, and will in general not affect the smoothness of the simulation for all but possibly the audited party.
- The protocol is designed to be integrated with existing middleware approaches with tolerable overhead. In particular, the protocol is abstract [11], i.e., it can be adapted for any specific MMOG.

Although the effort to integrate the protocol within a newly designed NVE is reasonable, the complication is to integrate the protocol into an existing code for a MMOG. A full blown MMOG is estimated to take $50 million prior to its launch [12] and therefore such an implementation is not accessible. Thus, we do not present a prototypical implementation.

Related Work on Security. Audit trails were successfully applied in electronic commerce applications (e.g., see [13]). An audit trail enables a special party, called *auditor*, to verify the correctness of previous transactions. The audit trail can either be stored at the client or the server. In any case, the audit information must be protected from modifications. Bellare and Yee [14] identified *forward security* as the key security property for audit trails: even if an attacker completely compromises the auditing system, he should not be able to forge audit information referring to the past. Implementations of secure audit and logging facilities can be found in [15, 16, 17].

The protocols described in this paper follow the principles of audit trails, but account for the specific particularities of NVE environments. Most importantly, our solution incurs a minimal network traffic overhead, while retaining its security. In fact, a direct adoption of classical audit trails to the NVE scenario would inflict a large load on the network, as the concrete state updates of all clients must be verified. In our solution, the audit information is stored at the client side and sent to the auditor on request. The client only "commits" itself to a status update by sending a short message to the server, which cannot be altered later.

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 [18].

Replication techniques for Byzantine fault tolerance [19, 20] also seem applicable to our problem. However, since the client has complete control of the replicas, these techniques cannot address the semantic-integrity problem caused by cheating on the client side. Still, replication based techniques are certainly applicable on the server side.

Only a few papers have dealt directly with security in online games. Baughman and Levine [21] and Chen and Maheswaran [22] concentrated on peer-to-peer multiplayer games, while we consider client server architectures of large-scale online games. Yan and Choi [23, 24] gave a taxonomy of security issues in online games and a case study on security of online bridge gaming. Davis [25] points out the importance of security in online games from a business perspective. Pritchard [26] deals with semantic attacks. However, this approach requires each client to run the entire simulation, which does not scale for MMOGS and is primarily targeted at small-scale peer-to-peer gaming applications.

2 Threat Analysis

The first NVEs were military simulation systems where the users belonged to well-defined groups whose NVE-clients were trusted. However, as large-scale NVES with untrusted and dispersed participants are becoming more popular, the security of NVEs becomes an eminent issue. We have identified the following security threats in the context of an NVE with untrusted participants:

1. **System Security Attacks:** There are a number of classical security problems associated with NVEs, such as authentication, or host security. These security issues have been widely studied [27, 28].
2. **Semantic Subversion:** The participants of an NVE can interact in the virtual environment according to the set of rules embodied in the simulation algorithms. The enforcement of these rules is of crucial importance for all honest participants and the system's host. We call attacks targeted at circumventing or subverting these rules semantic attacks.
    (a) **Semantic Integrity Violation:** Attacks in this category attempt to violate the physical and logical laws of the NVE without detection by the server. All attacks in this class involve maliciously modified software on the client side and come in two flavors:
        i. **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.
        ii. **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.
    (b) **Client Amplification:** In this case, the client employs modified software to achieve capabilities to exploit the possibilities of the NVE in an unintended manner. During such an attack, the externally observable behavior of the amplified client is not reliably distinguishable from the behavior of a honest client. Amplification attacks contain the following two main categories:
        i. **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, thus revealing a monster in a neighboring room that should have been hidden. Note that cheats of this kind may not require modifications to the client application — access to client memory or system libraries suffices for a cheat.

ii. **Agents:** The malicious client enhances the natural capabilities of the human participant. For example, an agent could automatically maintain a model of the world and employ search strategies to guide the player, or could log and replay successful prior actions.

3. **Metastrategies:** Attacks in this category are compliant with the NVE and do not involve software modifications. They exploit principal vulnerabilities present in the NVE, e.g., collusive collaboration of human participants, or mobbing of fellow participants.

Note that system security attacks are targeted against the server systems, while all other attack groups identified in this section describe exploits which involve only the client side.

System security attacks are exploits that do not involve specific properties of NVEs and therefore they are not in the scope of this paper. On the other extreme, Metastrategies do not violate the semantic rules of the game, and require solutions that look outside the environment. Consequently, the focus of this paper is on Semantic Subversion Attacks; these attacks are further subdivided into the categories Semantic Integrity Violation and Client Amplification. Some client amplification attacks can be addressed with memory encryption or other countermeasures [26], but not all can be handled in a rigorous way because they require models of human player capabilities. They are, however, amenable to statistical detection and countermeasures similar to intrusion detection systems [29].

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 conditions on the client behavior:

- **Rule Compliance:** Each client is only allowed to act in accordance with the rules of the NVE. This prevents rule corruption.
- **Monotone History:** The actions of the client must be irrevocable and undeniable. Consequently, clients are not allowed to choose an alternative history of actions once they obtain more information in the future. 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. We write ASTATE to denote the centrally maintained and abstracted state. Depending on the spatial position of Client_i within the simulated world, only a portion of the entire state is relevant for the Client_i; this portion is denoted by $\text{ASTATE}[\mathsf{Client}_i]$. 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 by the client.

Given an abstract state s, we use $\gamma(s)$ to denote the set of possible concretizations. Furthermore, if S is a concrete state, then $\alpha(S)$ is the unique abstract state which corresponds to S. The pair $\alpha()/\gamma()$ can be naturally viewed as a Galois connection between the set of abstract and concrete states [11], i.e., $S \in \gamma(\alpha(S))$ and $s = \alpha(S)$ for any $S \in \gamma(s)$.

When connecting to the NVE, Client_i receives a concrete state $S \in \gamma(\text{ASTATE}[\mathsf{Client}_i])$ to initialize its local state $\text{STATE}[\mathsf{Client}_i]$. From this point on, Client_i maintains and updates $\text{STATE}[\mathsf{Client}_i]$ locally and only receives abstract updates.

If Client_i wishes to change its state, it has to inform the $\mathsf{StateServer}$ in order to update the central NVE-state ASTATE. For this purpose, Client_i computes a state update in the form of a compact description Δ of the difference between the current state $\text{STATE}[\mathsf{Client}_i]_t$ and the intended next state; we call Δ a *diff.* Given a state S and a diff Δ between S and S', we denote the application of Δ to S by $S' = S \boxplus \Delta$. Note that Δ will typically be small compared to the state descriptions S if the NVE performs a fine-grained simulation of the virtual world.

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

One client cycle consists of the following steps: The client sends an abstraction of Δ, denoted by $\delta = \alpha(\Delta)$, to StateServer, which evaluates the semantics of the update. This abstract diff δ contains the changes requested by the client. We call δ a *request diff.* Now, two cases can happen:

- If δ is allowed with respect to the semantics of the NVE, then StateServer responds with a δ' that contains all changes intended by Client_i together with state updates performed by other clients present in the NVE. Upon receipt of δ', Client_i computes a concretization $\Delta' \in \gamma(\text{STATE}[\mathsf{Client}_i], \delta')$ and updates its own state by computing $\text{STATE}[\mathsf{Client}_i]_{t+1} = \text{STATE}[\mathsf{Client}_i]_t \boxplus \Delta'$.
- If δ is not consistent with the semantics of the NVE, the server refuses to apply the update. This can happen if Client_i tries to do something impossible, such as opening a locked door. Moreover, inconsistencies can be caused by synchronization and communication errors (e.g., packet loss of the network). In this case, the StateServer responds with a state update δ' which only contains the states updates of other clients.

In either case, 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 both the state of the clients and the server. However, if malicious clients participate in the simulation, this protocol is susceptible to *semantic integrity* violation as described in Section 1: StateServer is only able to check whether the *abstract* state updates $\delta = \alpha(\Delta)$ are consistent with its *abstract* state. A malicious Client (see Definition 5.3) can make an inconsistent state change Δ whose abstraction δ is consistent with the NVE rules.

4 Secure Semantic Integrity Protocol (SSIP)

In this section, we show how to amend the basic state update protocol described above with cryptographic mechanisms in order to prevent semantic integrity violation attacks. This protocol will satisfy all requirements established in Section 1.

Our approach is based on an audit procedure, which is performed by a dedicated server, namely the AuditServer. During each client cycle, the client sends a piece of evidence (containing a hash of the applied *concrete* state update) as action commitment to AuditServer. From time to time, the Client will also be requested to commit to a concrete state; these states will serve as possible starting states for the audit process.

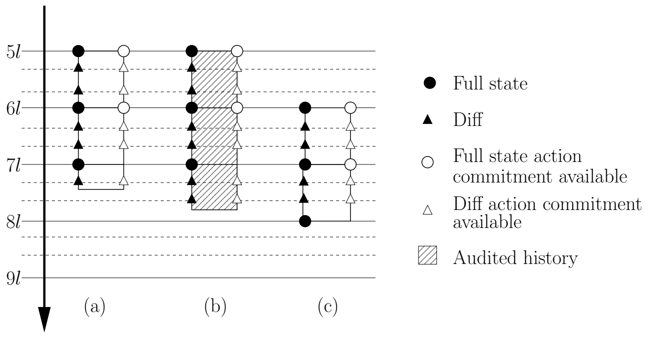
Note that our security model assumes that AuditServer is fully trusted. In particular, the protocols do not provide non-repudiation: a cheating audit server could frame innocent clients by wrongly claiming that they behaved badly. However, we do not consider this case, as we believe that it is not a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

When auditing is initiated, AuditServer asks a Client 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 Client computation and checks both its compliance to the NVE rules and its consistency with the action commitments sent previously.

Audits are initiated according to a strategy determined by the server, which is unpredictable for the client. For example, AuditServer might choose clients for auditing in a completely random fashion or audit "on demand" whenever statistical evidence suggests cheating.

Audit Cycles. The auditing process is organized in terms of *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. However, it is possible to customize l for each client while the simulation is running. The parameter l essentially determines how far back into the past auditing is possible.

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). In addition, as noted above, during each client cycle, the client sends 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.



5l
6l
7l
8l
9l
(a)
(b)
(c)
● Full state
▲ Diff
○ Full state action commitment available
△ Diff action commitment available
▨ Audited history

Figure 1. The "sliding window".

While StateServer only keeps the current abstracted central state, the clients do not only maintain their current concrete state but also *retain a history of previous states in a local buffer,* containing up to 3 full states and $3l$ diffs. In particular, the Client 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 deleted safely. 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

$$t_a = \left\lfloor \frac{t_0}{l} - 2 \right\rfloor l. \tag{1}$$

Thus, t_a denotes the expiration time for client side audit information. In addition to the history of states, the client stores all messages received from the server within the time interval determined by the sliding window.

Figures 1(a), (b), and (c) show the gradual change of the buffer of one specific client. These figures illustrate the buffer contents at client cycles $7l + 1$, $7l + 2$, and $7l + 3 = 8l$, respectively. The symbol $\bullet$ represents a fully saved state, whereas $\blacktriangle$ represents a concrete diff, both saved at the client. On the other hand, $\circ$ and $\triangle$ represent action commitments of full states and diffs which are available at the AuditServer (note that full state action commitments $\circ$ are only available after an audit cycle is finished).

As seen in Figure 1, at most three fully saved states are retained at any given time; the scope of an audit process covers at most three audit cycles (see Figure 1(b)). Once a new audit cycle is completed, the information about an earlier audit cycle can be discarded (see Figure 1(c)).

Audit Process. During the audit process, AuditServer requires Client to prove that its actions during the last two completely finished audit cycles and the current audit cycle are compliant to the rules of the NVE. For this purpose, Client sends the contents of the current sliding window (i.e., the state information) together with all corresponding state server messages to AuditServer, who simulates the client computations.

Now, AuditServer checks whether

1. the received state information matches the action commitments previously submitted by the client, whether

2. the client computation is compliant to the rules of the NVE and whether
3. 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 within the current client cycle, whereas action commitments of full states must be available only when the current audit cycle is completed. In Figure 1 the action commitments (represented by $\triangle$) for diffs are available at the AuditServer immediately. In contrast, the action commitment $\circ$ 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 guaranteed. In principle, the protocols can easily be adapted in such a way that more audit cycles are verified during each invocation of the audit protocol. However, for the sake of brevity, we present the protocols for the simplest case of auditing at most three audit cycles.

Protocol Description

Secure integrity enforcement is performed by three protocols ***Initialize***, ***StatusUpdate*** and ***Audit***. The protocol ***Initialize*** is performed whenever a client wants to join the NVE, whereas ***StatusUpdate*** is executed at each client cycle (i.e., whenever a client wishes to change its state). Finally, ***Audit*** implements the auditing mechanism. We assume that a client leaving the NVE performs an ordinary status update, where the diff encodes the intention to leave the NVE.

For the sake of simplicity, we present the protocol for a single client Client 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 a formal definition of these primitives, see [30]). For computing MAC-tags, an appropriate key must be generated with $k = \text{GenMac}(1^n)$ 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 $\text{VerifyMac}(k, m, t) = \{\text{true}, \text{false}\}$. We write $M = \text{AuthMsg}(k, m, \mathsf{Client})$ as an abbreviation for $m \,\|\, \text{SignMac}(k, m \,\|\, \mathsf{Client})$, 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 $M^{(2)} = \text{SignMac}(k, m \,\|\, \mathsf{Client})$. 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 $\text{STATE}[\mathsf{Client}]_t$ 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 Client. The MAC enables AuditServer to check whether a cheating Client has passed modified status update messages to the AuditServer.

In the following, we describe each protocol in detail:

***Initialize*:** This protocol initializes the state of a Client wishing to join the NVE (see Figure 2).

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

Figure 2. Protocol *Initialize*

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

Figure 3. Protocol *StatusUpdate*

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. In a practical implementation, the index of the hash function would be fixed and globally 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 $S \in \gamma(\text{ASTATE}[\mathsf{Client}])$ to the client. The client initializes its local state S_0 with S. Note that 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, we only require that this process is completed before the lth client cycle is initiated.

***StatusUpdate*:** After initialization, the client uses this protocol to update its local state in each client cycle to reflect actions of the client itself, of other clients, and the state server. Formally, the protocol is shown in 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 sends an abstracted request diff $\delta_{t+1} = \alpha(\Delta_{t+1})$ to StateServer. The server checks whether this request is valid and consistent with the current central NVE state ASTATE and computes a new

1. AuditServer $\rightsquigarrow$ Client : $\quad$ *audit* $\parallel t_0$
2. Client computes $t_a = \left\lfloor \frac{t_0}{l} - 2 \right\rfloor l$
3. Client $\rightsquigarrow$ AuditServer : $\quad S_{t_a} \parallel \Delta'_{t_a+1} \parallel \ldots \parallel \Delta'_{t_0} \parallel M_{t_a+1} \parallel \ldots \parallel M_{t_0}$
4. AuditServer computes $\hat{S}_{i+1} = \hat{S}_i \boxplus \Delta'_{i+1}$ 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$(k, M_i^{(1)} \parallel \text{Client}, M_i^{(2)}) =$ 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 $t_a = 0$, Client $\rightsquigarrow$ AuditServer : $M_0^{(2)}$ and AuditServer checks VerifyMac$(k, S_0 \| \text{Client}, M_0^{(2)}) =$ true.
8. AuditServer accepts the computations of Client if and only if all tests in steps 5 to 7 passed.

Figure 4. Protocol *Audit*

authoritative diff δ'_{t+1}. This diff might differ from δ_{t+1} since it has to reflect changes of other clients and the server itself; however, if δ_{t+1} is legitimate with respect to the NVE semantics, δ'_{t+1} contains the state changes of δ_{t+1}. If δ_{t+1} violates the semantic integrity, δ'_{t+1} *only* contains the state updates of the other clients but *not* δ_{t+1} (or at most those actions in δ_{t+1} that are consistent). The StateServer updates its centrally managed state ASTATE according to δ'_{t+1} and sends δ'_{t+1} back to the client, together with a MAC and an incremented nonce (steps 1-4 of the protocol).

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

For audit purposes, the client saves all information as evidence that is necessary for the audit server to simulate its computations. More precisely, at the beginning of each audit cycle, the client saves its full state; in intermediate client cycles, the client only retains diffs to the previous state. In addition, the client saves all messages M_i received from the state server. Finally, all outdated audit information (i.e., the fully saved state, all diffs and messages belonging to the third-last audit cycle) can be removed (step 8).

***Audit*:** During the audit protocol, AuditServer validates the computations of one Client. 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 (see Figure 4).

The auditing protocol starts with an audit message sent to the Client during client cycle t_0. An audit can be initiated at any client cycle $t_0 \geq 2l$. The client first computes the starting point t_a of the audit according to equation (1). The client then sends the concrete state S_{t_a} as well as all diffs Δ'_i and messages M_i for $t_a + 1 \leq i \leq t_0$ to the AuditServer (steps 1-3 of the protocol). Finally, the audit server checks, using the action commitment messages D_i and Q_i submitted by the client before, whether the client adhered to the NVE semantics. In particular, the audit server checks

- whether all Δ_i' are suitable concretizations of δ_i' sent by the state server in message M_i (step 5),
- whether all state server messages M_i ($t_a + 1 \leq i \leq t_0$) are unmodified (step 6(a)) and
- whether all action commitment messages (D_t and Q_t) submitted by the client beforehand are valid, in particular the AuditServer checks
  - the hashes on the messages D_i, $t_a + 1 \leq i \leq t_0$, (step 6(b)) 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} (step 7). Note that these messages are already available to the audit server if the timing conditions of the ***StatusUpdate*** protocol are enforced.
- If the first audit cycle is to be audited ($t_a = 0$), then Client is required to present $M_0^{(2)} = \mathrm{SignMac}(k, S_0 \| \mathsf{Client})$ to AuditServer additionally to prove that the initial state S_0 has been authorized by the StateServer (step 7).

If all checks pass, the client is considered honest (step 8).

Computation and Engineering Overhead. The protocol can be implemented in a very resource efficient manner: The ***StatusUpdate*** protocol requires only a few MAC and hash computations over relatively small amounts of data. The hash computation over the complete state of a client can be processed in background during the l client cycles of an audit cycle. Moreover, only the MACs and hashes are additionally transmitted over the network.

In contrast, the ***Audit*** protocol is much more data intensive and involves a complete re-simulation of the client computations. However, the execution of the ***Audit*** protocol is not time critical and can be delegated to a specific server, namely the AuditServer. Therefore, it does not cause resource overhead at the StateServer.

To integrate our protocol into an NVE system, one has to implement the protocol logic, add the computation of the MACs and hashes at the state server and the client, and implement the audit server. It should be possible to implement the audit server by mainly reusing client code since the audit server is simulating the client computations. The biggest problem in implementing the protocol will likely be the creation of copies of the complete client state in a timely manner, as required at the beginning of each audit cycle. However, all remaining parts of the protocol can be implemented in a straight forward manner.

5 Security

In this section, we will state the security property achieved by the Secure Semantic Integrity Protocol SSIP. In particular, we introduce two properties (namely *rule compliance* and *monotone history,* discussed briefly in Section 2) which jointly assure the semantic integrity of the NVE. We show that the protocol above enforces both properties.

We introduce the *successor relation* $\succ$, where $\textsc{state} \succ \textsc{state}'$ holds if there is a diff Δ such that $\textsc{state}' = \textsc{state} \boxplus \Delta$. Analogously, $\textsc{astate} \succ \textsc{astate}'$ is true, if there is an abstract diff δ such that $\textsc{astate}' = \textsc{astate} \boxplus \delta$. Since by definition $\textsc{state}' = \textsc{state} \boxplus \Delta$ implies $\alpha(\textsc{state}') = \alpha(\textsc{state}) \boxplus \alpha(\Delta)$, we find that $\textsc{state} \succ \textsc{state}' \Rightarrow \alpha(\textsc{state}) \succ \alpha(\textsc{state}')$ holds, see Section 3. However, the converse is not necessarily true, since it is possible that there are no concrete states which realize a given abstract transition.

Based on the successor relation, we define a sequence of concrete states $\langle \textsc{state}_0, \ldots, \textsc{state}_t \rangle$ as *valid* if $\forall\ 0 \leq i < t\ :\ \textsc{state}_i \succ \textsc{state}_{i+1}$ holds. Analogously, $\langle \textsc{astate}_0, \ldots, \textsc{astate}_t \rangle$ is a *valid* sequence of abstract states if $\forall\ 0 \leq i < t\ :\ \textsc{astate}_i \succ \textsc{astate}_{i+1}$.

During the ***Initialize***- and ***StatusUpdate***-protocol, the client receives a concrete state STATE_0 and a series $\langle \delta'_1, \ldots, \delta'_t \rangle$ of (abstract) authoritative diffs. Thus, the client and the StateServer *produce* cooperatively a sequence of abstract states $\langle \text{ASTATE}_0, \ldots, \text{ASTATE}_t \rangle$, where $\text{ASTATE}_0 = \alpha(\text{STATE}_0)$ and $\text{ASTATE}_{i+1} = \text{ASTATE}_i \boxplus \delta'_{i+1}$. The StateServer only checks whether the abstract sequence $\langle \text{ASTATE}_0, \ldots, \text{ASTATE}_t \rangle$ is valid; as noted Section 1, the concept of abstract states has been introduced to relieve StateServer from the workload of maintaining the details of the concrete representation. But the validity of the abstract sequence does not guarantee its *realizability:* We say that $\langle \text{ASTATE}_0, \ldots, \text{ASTATE}_t \rangle$ is *realizable at* concrete state STATE_0, if there exists a valid concrete sequence $\langle \text{STATE}_0, \ldots, \text{STATE}_t \rangle$ which starts with STATE_0 and where $\text{STATE}_i \in \gamma(\text{ASTATE}_i)$ holds for all $0 \leq i \leq t$.

Definition 5.1 Rule Compliance
A client behaves rule compliant, *if the sequence of abstract states* $\langle \text{ASTATE}_0, \ldots, \text{ASTATE}_t \rangle$ *produced by the client and the* StateServer *is realizable at concrete state* STATE_0. •

During the ***Audit***-protocol, the AuditServer asks the client to disclose some of its former concrete states $\langle \text{STATE}_0, \ldots, \text{STATE}_t \rangle$. In this situation, it may happen that the sequence disclosed by the client is rule compliant, but the client has manipulated this sequence in order to make its history look as if it were compliant to the rules of the NVE. In Section 2, we called such a behavior causality alteration. For a given $q \geq t$, we denote by HSTATE_t^q the state returned by the client if queried at time q for its former state STATE_t. HSTATE_t^q is a *historical state.*

If the client is honest, it will always return the truthful historical states, i.e., $\text{HSTATE}_t^q = \text{STATE}_t$ for all $q \geq t$. In this case, the client never rewrites its history (i.e., $\text{HSTATE}_t^q \neq \text{HSTATE}_t^r$) and thus we say the client has a monotone history.

Definition 5.2 Monotone History
A client has a monotone history, if $\text{HSTATE}_t^q = \text{STATE}_t$ *for all* $q \geq t$. •

A client which obeys the requirements of rule compliance and monotone history is a honest client.

Definition 5.3 Honest and Malicious Clients
A client is honest, if it behaves rule compliant and discloses a monotone history. Otherwise, the client is malicious. •

The following theorem show that (under standard cryptographic assumptions) the protocol of Section 4 enforces honest client behavior.

Theorem 5.4 Security of the Secure Semantic Integrity Protocol (SSIP)
If CFHash is a collection of collision-free hash functions, and SignMac is a message authentication code secure against selective forgery of messages, then the Secure Semantic Integrity Protocol (SSIP) enforces honest client behavior(assuming probabilistic polynomial time computations on client- and server-side).

We show the theorem by a simultaneous reduction to the problems of finding collisions of the hash function CFHash and of forging MACs of SignMac. Suppose that there is a client that succeeds to cheat with non-negligible probability by violating either the security property of rule compliance or of monotone history. Then, we show that there exist algorithms AttackHash and AttackMac that can compute collisions of CFHash or forge MACs; one of the algorithms will have non-negligible success probability, which violates the assumptions.

As a proof technique, we use the concept of a *malicious behavior experiment,* consisting of a ScenarioGenerator, which simulates both a StateServer and a set of clients. A ScenarioGenerator is designed for a specific malicious client and provides this client with a view which is identical to a real-world execution of the NVE. The simulation, as provided by the ScenarioGenerator, is designed to allow the corresponding client to behave maliciously, i.e., the ScenarioGenerator is intuitively used to demonstrate the malicious behavior of the client.

At some point during the simulation, an audit process is initiated and the malicious client responds to the request accordingly. We say that the malicious behavior experiment is *successful,* if the client could cheat within the audited client cycles such that the involved AuditServer is unable to detect the malicious behavior.

By a case analysis, we show that, whenever a malicious client is able to cheat the AuditServer in a malicious behavior experiment, it is possible to find either a collision of the hash function or a forged MAC: If there is a client together with a ScenarioGenerator which is performs successfully in malicious behavior experiments with non-negligible probability, then we can either produce collisions of the hash function or forged MACs with non-negligible probability.

Thus, we show that the existence of a client which is successful in the malicious code experiment violates standard cryptographic assumptions. Therefore, the SSIP enforces honest client behavior. For the detailed proof see Appendix A.

6 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] Epic Games. Unreal tournament. `http://www.unrealtournament.com`, 1999.

[2] Linden Lab. Second life. `http://secondlife.com`, 2003.

[3] Virtual online world. The Economist, September 28 2006. `http://www.economist.com/business/displaystory.cfm?story_id=E1_SJGPVPR`.

[4] CCP Games. Eve online. `http://www.eve-online.com`, 2003.

[5] Blizzard. World of warcraft. `http://www.worldofwarcraft.com`, 2005.

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

[7] A. Pasick. Us congress launches probe into virtual economies, October 15 2006. `http://secondlife.reuters.com/stories/2006/10/15/us-congress-launchs-probe-into-virtual-economies/`.

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

[9] C. Joslin, T. Di Giacomo, and N. Magnenat-Thalmann. Collaborative virtual environments: Form birth to standardization. *IEEE Communications*, pages 28–33, April 2004.

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

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[12] Michael Zenke. Interview: Call of the wild - turbine's jessica mulligan on MMO trends. `http://www.gamasutra.com/features/20050422/zenke_01.shtml`, 2005.

[13] J. Peha. Electronic commerce with verifiable audit trails. In *Proceedings of INET'99, Internet Society*, 1999.

[14] M. Bellare and B. Yee. Forward integrity for secure audit logs. Technical report, UCSD, 1997.

[15] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and Systems Security*, 2(2):159–176, 1999.

[16] B. Schneier and J. Kelsey. Remote auditing of software outputs using a trusted coprocessor. *Future Generation Computer Systems*, 13(1):9–18, 1997.

[17] C. Chong, Z. Peng, and P. Hartel. Secure audit logging with tamper-resistant hardware. In *Proceedings of the 18th IFIP International Information Security Conference*, pages 74–84, 2003.

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

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

[20] M.K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *Proceedings of the ACM Conference on Computer and Communications Security*, 1994.

[21] N. Baughman and B. Levine. Cheat-proof playout for centralized and distributed online games. In *Proceedings of the 20th IEEE INFOCOM Conference*, pages 104–113, 2001.

[22] B. Chen and M. Maheswaran. A fair synchronization protocol with cheat proofing for decentralized online multiplayer games. In *Proceedings of the 3rd IEEE Symposium on Network Computing and Applications*, pages 372–375, 2004.

[23] J. Yan. Security issues in online games. *The Electronic Library: international journal for the application of technology in information environments*, 20(2), 2002.

[24] J. Yan and H.J. Choi. Security design in online games. In *Annual Computer Security Applications Conference*, 2003.

[25] S. Davis. Why cheating matters. cheating, game security and the future of on-line gaming business. In *Game Developers Conference*, 2003.

[26] M. Pritchard. How to hurt the hackers: The scoop on the internet cheating and how you can combat it. *Game Developer Magazine*, June 2000. `http://www.gamasutra.com/features/20000724/pritchard_01.htm`.

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

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

[29] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(9):805–822, 1999.

[30] O. Goldreich. *Foundations of Cryptography*, volume II: Basic Applications. Cambridge University Press, 2004.

Appendix

A Proof of the Main Theorem

In order to prove Theorem 5.4, we show that whenever a client is able to behave maliciously with non-negligible probability, either a MAC-tag can be forged or a collision of the hash function can be found in probabilistic polynomial time, again with non-negligible probability.

We naturally assume that AuditServer executes a single step of the ***StatusUpdate***- and the ***Audit***-protocol within polynomial time with respect to the security parameter n and the size of the concrete states.

As a malicious client can only operate in the environment of an NVE, we will use a scenario generator in our proof which provides a client with a realistic environment by pretending to be the state server. The client interacts with the scenario generator in the same way as it would with a state server.

Definition A.1 Scenario Generator
ScenarioGenerator is a probabilistic interactive Turing Machine which takes the security parameter 1^n as initial input and produces a scenario $\langle S_0, \delta'_1, \dots, \delta'_m \rangle$ in the following way:

- *First, it outputs a concrete state S_0. After this initial computation, the ScenarioGenerator repeats the following: it waits for an abstract request diff δ_t from the client and returns a corresponding authoritative diff δ'_t for $0 < t \leq m$ where m is the length of the scenario; m must be bounded by a polynomial in n.*
- *The computation of the initial concrete state S_0 and the computation of each abstract diff δ_t must be done in probabilistic polynomial time.* •

The ultimate goal of the client, while interacting with the ScenarioGenerator, is to behave maliciously while being undetected by the AuditServer. More precisely, the allegedly malicious client, the ScenarioGenerator, and the trusted real-life AuditServer engage in the following experiment in which the ScenarioGenerator behaves in accordance with Definition A.1.

Definition A.2 Malicious Behavior Experiment
*In the malicious behavior experiment, a ScenarioGenerator, a Client, and the AuditServer participate. First, the security parameter n is distributed and the AuditServer communicates the index h of the hash function to be used by the client. Also, the ScenarioGenerator sends an initial state S_0, utilizing the **Initialize**-protocol, to the Client. Then ScenarioGenerator and Client repeatedly execute the **StatusUpdate**-protocol for at most m rounds, where m is the length of the generated scenario. In each round, Client sends a request diff δ_i to ScenarioGenerator and receives an authoritative diff δ'_i as response. Furthermore, the client outputs its currently locally maintained state S_i in each iteration.*

*The AuditServer initiates the **Audit**-protocol once during the experiment at a uniformly and randomly chosen point in time $t_0 \leq m$. The experiment is successful, if the client behaves maliciously within the audited time frame but is not detected.* •

Figure 5 depicts this experiment graphically: GenMac is used to generate a random MAC-key which will subsequently used by AuthMsg to authorize the messages originating from the ScenarioGenerator. The ScenarioGenerator produces the initial state S_0 and the authoritative updates $\delta'_1, \dots, \delta'_m$, and authenticates them with the help of an oracle AuthMsg, similarly as the StateServer. The authenticated messages $M_0, \dots, M_m$ are sent to Client as usual in the ***StatusUpdate***-protocol.

As in the real protocol execution, AuditServer receives during the executions of the ***StatusUpdate***-protocol the hashes $D_1, \dots, D_m$. Similarly, the Client sends to AuditServer the hashes $Q_0, \dots, Q_{\lceil \frac{m}{l} \rceil l}$ on the entire state and produces the sequence of local states $S_0, \ldots, S_m$. In the figure, we use thin lines to depict messages which are sent either during the ***Initialize***- or ***StatusUpdate***-protocol.

In contrast, the bold lines are used for messages of the ***Audit***-protocol: At some uniformly and randomly chosen point in time t_0, AuditServer initiates the ***Audit***-protocol by sending the message $audit \| t_0$ to the Client. In response, the client will reply with the messages $\bar{M}_{t_a+1}, \ldots, \bar{M}_{t_0}$, $\bar{S}_{t_a}$, and $\bar{\Delta}_{t_a+1}, \ldots, \bar{\Delta}_{t_0}$. By doing so, Client claims that $M_i = \bar{M}_i$, $S_{t_a} = \bar{S}_{t_a}$, and $\Delta_i = \bar{\Delta}_i$, where S_{t_a} and $\Delta_{t_a+1}, \ldots, \Delta_{t_0}$ denote states and diffs the client originally committed to.

The experiment ends when AuditServer outputs a verdict whether Client behaved maliciously or not.

In Theorem 5.4, we said that the SSIP *enforces honest client behavior,* assuming polynomial time complexity bounds on the server and client side and assuming that MACs and collections of collision-free hash functions do exist. With the definition of the malicious behavior experiment at hand, we are able to refine this statement as follows: We prove that a client which passes the malicious behavior experiment successfully with a non-negligible probability will yield either a procedure AttackHash which produces collisions for the allegedly secure hash function or a procedure AttackMac which forges MACs for the allegedly secure MAC function. At least one of these procedures will be successful with non-negligible probability, assuming that the malicious behavior experiment succeeds with non-negligible probability. Using this terminology, we can formulate the theorem as follows:

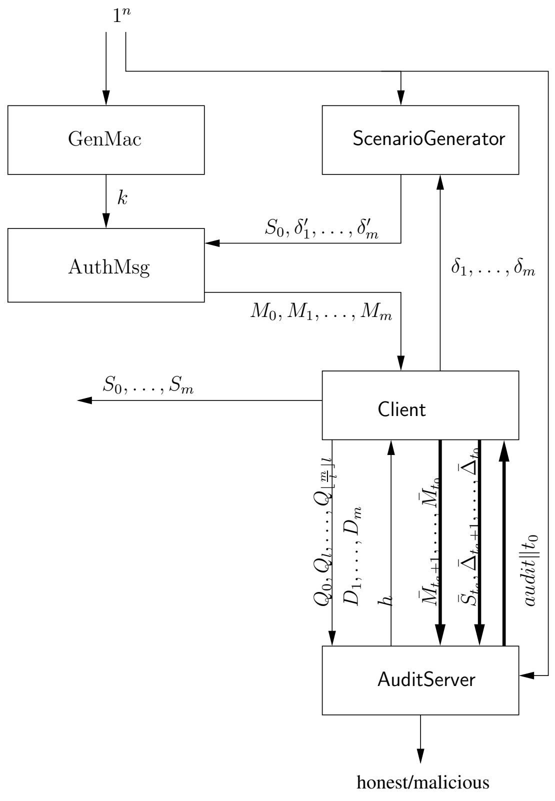

1^n
GenMac
ScenarioGenerator
k
$S_0, \delta'_1, \ldots, \delta'_m$
AuthMsg
$\delta_1, \ldots, \delta_m$
$M_0, M_1, \ldots, M_m$
$S_0, \ldots, S_m$
Client
$Q_0, Q_l, \ldots, Q_{\lfloor \frac{m}{l} \rfloor l}$
$D_1, \ldots, D_m$
h
$\bar{M}_{t_a+1}, \ldots, \bar{M}_{t_0}$
$\bar{S}_{t_a}, \bar{\Delta}_{t_a+1}, \ldots, \bar{\Delta}_{t_0}$
$audit \| t_0$
AuditServer
honest/malicious

Figure 5. Malicious Behavior Experiment

Theorem A.3 Security of the Secure Semantic Integrity Protocol (SSIP) (II)
If CFHash is a collision-free hashing function and SignMac a message authentication code, then the Secure Semantic Integrity Protocol (SSIP) guarantees that any malicious behavior experiment with a probabilistic polynomial-time client has negligible success probability. •

Proof of Theorem A.3
Suppose there exist a Client and a ScenarioGenerator such that the client succeeds in the malicious behavior experiment with non-negligible probability.

Then we construct two probabilistic polynomial-time procedures where the first procedure AttackHash is able to find a collision of the hash function CFHash, and the second procedure AttackMac is able to forge MACs. Either of these algorithms will succeed with non-negligible probability, contradicting the cryptographic assumptions of Theorem A.3.

Both procedures are constructed on the basis of a malicious behavior experiment. In particular, we claim that each successful execution of the experiment yields either a forged MAC or a collision of the hash function.

According to Definition 5.3, a successful malicious client must either violate the rule compliance of the NVE or the monotone history property, while being undetected. The rule compliance property is violated if the sequence of abstract states, as presented by the Client to the AuditServer, is not jointly produced by the StateServer and the Client or is not realizable at the given initial concrete client state. A cheating client can attempt to present a different sequence by manipulating the received authoritative diffs δ'_i. The realizability of the sequence is always checked correctly in the ***Audit***-protocol by resimulating the presented concrete sequence and checking whether it corresponds to the abstract one.

The monotone history property is violated, if the client provides the AuditServer with a sequence of states that differs from the sequence of states it actually executed. Thus, the client may either cheat on the diffs $\bar{\Delta}_i$ sent during the audit or on the full state $\bar{S}_{t_a}$ that provides the basis for the auditing process. In summary, the malicious client has the following options to cheat:

1. The client cheats on the first audited state S_{t_a}, i.e., the state $\bar{S}_{t_a}$ sent to the AuditServer differs from S_{t_a}. In this case the client has found a second preimage of the hash function, as $\text{CFHash}_h(S_{t_a}) = \text{CFHash}_h(\bar{S}_{t_a})$.

2. Suppose now that the client does not cheat on the first audited state, i.e., $\bar{S}_{t_a} = S_{t_a}$. Thus, the client may either cheat on some message M_i or honestly report $\bar{M}_i = M_i$ for all i.

   (a) In the first case, the client must provide a message $\bar{M}_i = \text{AuthMsg}(k, \bar{\delta}'_i \,\|\, n_i, \mathsf{Client})$ for a $\bar{\delta}'_i$ which has never been authenticated (since we assume that the malicious behavior experiment succeeds). Thus, the client would be able to forge the MAC of the message $\bar{\delta}'_i \,\|\, n_i \,\|\, \mathsf{Client}$, which has never been authenticated during the entire experiment due to the uniqueness of the nonce.

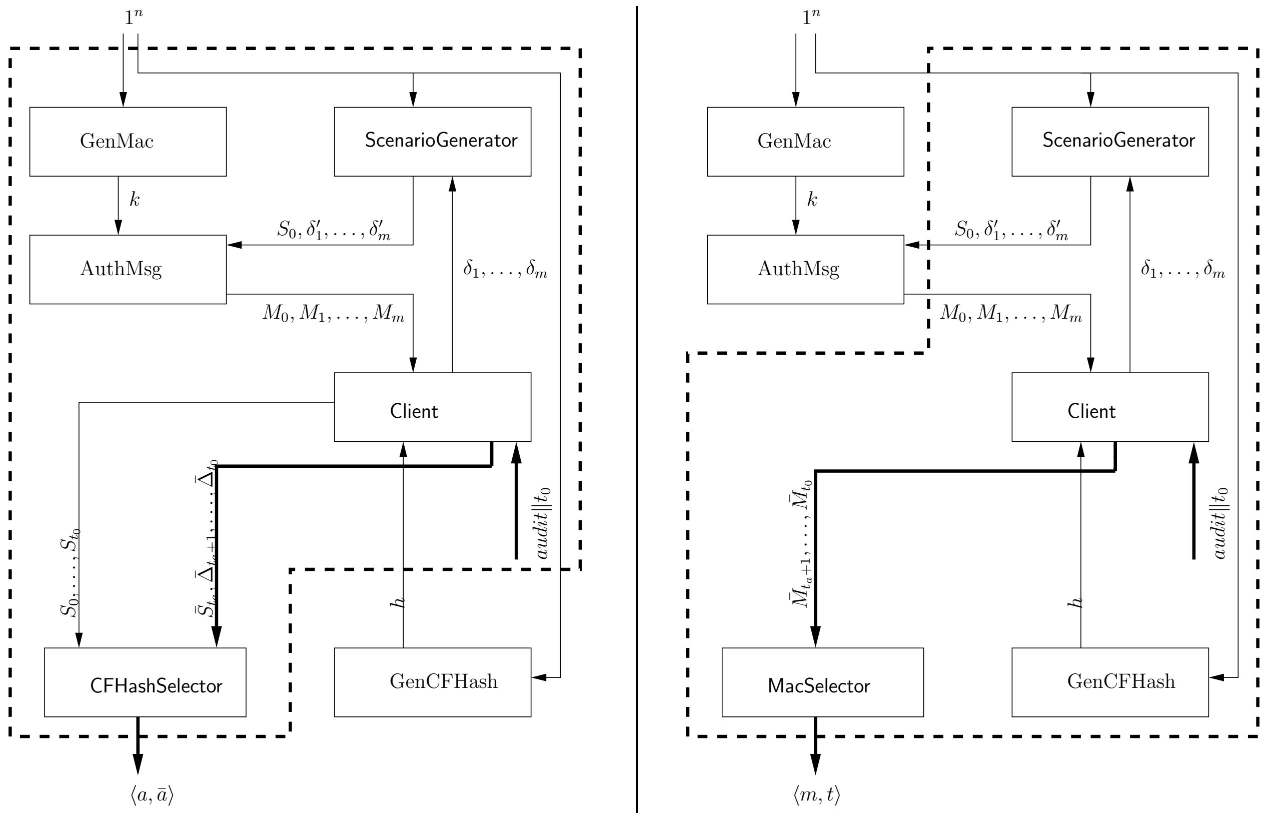
   (b) In the second case ($\bar{M}_i = M_i$ for all i), the client provided the AuditServer with the correctly authenticated authoritative diffs, as constructed by ScenarioGenerator. This leave the client with two other possibilities for cheating:
   Case 1: The client cheats on a diff, i.e., there is an i such that $\bar{\Delta}'_i \neq \Delta'_i$. Since the client committed to Δ'_i by sending $D_i = \text{CFHash}_h(\Delta'_i)$ to AuditServer, it follows that the client has found a second preimage of D_i.
   Case 2: If the client does not cheat on the diffs, the only remaining way to cheat successfully without being detected is to manipulate the state S_{t_a+l}. This means that S_{t_a+l} differs from $\bar{S}_{t_a+l} = \bar{S}_{t_a} \boxplus \bar{\Delta}_{t_a+1} \boxplus \ldots \boxplus \bar{\Delta}_{t_a+l}$. But the client already committed itself to S_{t_a+l} by sending the hash $Q_{t_a+l} = \text{CFHash}_h(S_{t_a+l})$ to the AuditServer. Since the client cheats undetected, it must have found a second pre-image to Q_{t_a+l}.

Since by assumption the experiment is successful with a non-negligible probability, we can either forge MACs or compute second pre-images of the collision-free hash function with non-negligible probability.

It remains to construct the two attack procedures AttackMac and AttackHash which adapt the black-box simulation and fit the definition of attacks on MACs and collision-free hashing functions, respectively.

- We build an attack procedure AttackHash as depicted in Figure 6a: The attack procedure AttackHash receives the index h of the collision function to be used and outputs a pair $\langle a, \bar{a} \rangle$ such that $\text{CFHash}_h(a) = \text{CFHash}_h(\bar{a})$ holds with non-negligible probability.

  GenMac and AuthMsg are part of the attack procedure, while GenCFHash is external to the attack. The AuditServer has been replaced: GenCFHash is used to provide the index h for the hash function and



1^n
GenMac
ScenarioGenerator
k
$S_0, \delta'_1, \ldots, \delta'_m$
AuthMsg
$\delta_1, \ldots, \delta_m$
$M_0, M_1, \ldots, M_m$
Client
$S_0, \ldots, S_{t_0}$
$\bar{S}_{t_a}, \bar{\Delta}_{t_a+1}, \ldots, \bar{\Delta}_{t_0}$
h
$audit \| t_0$
CFHashSelector
GenCFHash
$\langle a, \bar{a} \rangle$
1^n
GenMac
ScenarioGenerator
k
$S_0, \delta'_1, \ldots, \delta'_m$
AuthMsg
$\delta_1, \ldots, \delta_m$
$M_0, M_1, \ldots, M_m$
Client
$\bar{M}_{t_a+1}, \ldots, \bar{M}_{t_0}$
h
$audit \| t_0$
MacSelector
GenCFHash
$\langle m, t \rangle$

(a) AttackHash (b) AttackMac

Figure 6. Attack Procedures

the message $audit \| t_0$, which starts the ***Audit***-protocol, is sent at a uniformly and randomly chosen point in time.

The CFHashSelector computes the sequence $\Delta'_{t_a+1}, \ldots, \Delta'_{t_0}$ (based on $S_{t_a}, \ldots, S_{t_0}$) and the state $\bar{S}_{t_a+l}$ (based on the $\bar{S}_{t_a}$ and $\bar{\Delta}'_{t_a+1}, \ldots, \bar{\Delta}'_{t_0}$). Then CFHashSelector chooses uniformly and randomly one pair $\langle a, \bar{a} \rangle$ from the pairs $\left\langle S_{t_a}, \bar{S}_{t_a} \right\rangle$ and $\left\langle S_{t_a+l}, \bar{S}_{t_a+l} \right\rangle$ and from the sequence of pairs $\left\langle \Delta'_{t_a+1}, \bar{\Delta}'_{t_a+1} \right\rangle, \ldots, \left\langle \Delta'_{t_0}, \bar{\Delta}'_{t_0} \right\rangle$.

All other outputs of the Client, namely the authenticated messages $\bar{M}_{t_a+1}, \ldots, \bar{M}_{t_0}$, the hashes $D_1, \ldots, D_m$ on the concrete diffs, and the hashes $Q_0, \ldots, Q_{\lceil \frac{m}{l} \rceil l}$ on the entire states, are discarded.

Let us assume that the simulation produces a collision for the collision-free hashing function with non-negligible probability, i.e. that one of the possible choices for $\langle a, \bar{a} \rangle$ is a collision under the given hash function with non-negligible probability. Then the randomly chosen pair is still a collision with non-negligible probability.

The runtime of AttackHash is again polynomially bounded since all components of the malicious behavior experiment are running within probabilistic polynomial time.

- We construct an attack procedure AttackMac which takes 1^n as input, has access to an authentication oracle AuthMsg, and produces with non-negligible probability a pair $\langle m, t \rangle$ such that m has not been authenticated before by AuthMsg but such that $t = \mathrm{SignMac}(k, m)$ holds for a key k which is not known to the AttackMac.

In Figure 6b, AttackMac is shown as the procedure which is enclosed by the dashed box: This time, the GenMac and AuthMsg procedures are external to AttackMac such that the used key k is inaccessible to AttackMac (and again, we replace the AuditServer by GenCFHash and send the first message of the ***Audit***-protocol at a randomly chosen point in time).

The MacSelector receives the messages $\bar{M}_{t_a+1}, \ldots, \bar{M}_{t_0}$ and selects one of them uniformly and randomly. If one of these messages contains a forged MAC-tag with non-negligible probability, then a uniformly selected message $\bar{M}_i$ contains still a forged MAC-tag with non-negligible probability. This is true, since there are at most polynomially many such pairs. Finally, the procedure outputs $\left\langle \bar{M}_i^{(1)}, \bar{M}_i^{(2)} \right\rangle$.

All other outputs of the Client, namely the sequence of local stats $S_0, \ldots, S_{t_0}$, the allegedly occurred state at the beginning of the audited cycle $\bar{S}_{t_a}$ and the allegedly subsequently used diffs $\bar{\Delta}_{t_a+1}, \ldots, \bar{\Delta}_{t_0}$, the hashes $D_1, \ldots, D_m$ on the concrete diffs, and the hashes $Q_0, \ldots, Q_{\lceil \frac{m}{l} \rceil l}$ on the entire states, are discarded.

Since all components used within AttackMac run within polynomial time, ScenarioGenerator runs in polynomial time with respect to the security parameter n as well.

This concludes the proof: If the protocol does not eliminate the possibility of successful executions of the malicious behavior experiment with non-negligible probability, then it either produces forged MACs or collisions for the hash function with non-negligible probability, and therefore at least one of the procedures AttackMac and AttackHash will be successfully attacking the corresponding cryptographic primitive—which contradicts the assumption that no such attack exists at all. •