Semantic Integrity in Large-Scale Online Simulations

SOMESH JHA
University of Wisconsin
STEFAN KATZENBEISSER
Technische Universität Darmstadt
CHRISTIAN SCHALLHART
Technische Universität Darmstadt
HELMUT VEITH
Technische Universität Darmstadt
and
STEPHEN CHENNEY
Emergent Game Technology

As large-scale online simulations such as Second Life and World of Warcraft are gaining economic significance, there is a growing incentive for attacks against such simulation software. We focus on attacks against the semantic integrity of the simulation. This class of attacks exploits the client-server architecture and is specific to online simulations which, for performance reasons, have to delegate the detailed rendering of the simulated world to the clients. Attacks against semantic integrity often compromise the physical laws of the simulated world—enabling the user's simulation persona to fly, walk through walls, or to run faster than anybody else.

We introduce the Secure Semantic Integrity Protocol (SSIP) which enables the simulation provider to audit the client computations. Then we analyze the security and scalability of SSIP. First, we show that under standard cryptographic assumptions SSIP will detect semantic integrity attacks. Second, we analyze the network overhead, and determine the optimum tradeoff between cost of bandwidth and audit frequency for our protocol.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Security

Additional Key Words and Phrases: Networked Virtual Environments, cryptographic protocols, semantic integrity

Supported by the European FP6 project ECRYPT (IST-2002-507932).

1. INTRODUCTION

With the advent of *Second Life*, large-scale online simulations of virtual worlds have for the first time attracted broad public interest beyond the computer gaming community. It is widely expected that the technology of Networked Virtual Environments (NVEs) will provide a novel paradigm for the Internet which will partly replace the current web browser technology. Major industrial investments in NVE technology as well as virtual representations [Siklos 2006; Gardner 2007], standardization efforts for seamless transition between online worlds [Economist 2007b], increasing academic interest (special issues on NVEs were published by the IEEE [IEEE 2004] and ACM [ACM 2003; 2004]) and continuous press coverage are all witness to the impending technology leap.

Both in gaming and Second Life we have seen that virtual worlds are becoming fast-growing economies in their own right. There is a thriving market for virtual goods and territories, and virtual money can be converted to cash [Economist 2007a]. It is safe to assume that within a decade, virtual worlds will hold significant segments of the online shopping market.

Although the attractiveness of virtual worlds is often attributed to the possibility of bending certain physical laws—e.g., changing one's physical appearance or flying—the users expect the virtual world to differ from reality only in regular and predictable aspects, while the laws of physics and logic are by whole and large intact. Most importantly, the virtual world has to be consistent and fair, i.e., repeated online experiments should yield the same result regardless of the user. We will refer to this property of the virtual world as its *semantic integrity*. The semantic integrity of a virtual world is typically not formally defined, but an emergent property of the simulation software which combines physical simulations with game logic, animated semi-intelligent objects, etc.

Virtual economies are highly sensitive to violations of semantic integrity: in order to sustain a realistic and fair environment, most simulations will prohibit duplication of virtual objects, access to virtual media before purchase, walking through walls, walking on water etc. In order to establish a common understanding of the simulated world, it is particularly important to enforce identical policies and restrictions on all participants. By the same reasoning, however, there is a strong incentive for malicious users to manipulate the semantic integrity of the simulation in order to gain financial profit or other advantages.

The key to attacks against the semantic integrity of NVEs lies in their distributed software architecture. In a large-scale NVE, the central state server is maintaining an *abstract view of the virtual world*, in particular a simplified geometry, while all clients are responsible to maintain a *concrete* state. Thus the computational load of the detailed simulation is distributed among the clients and thus the bandwidth requirements can be reduced. With current technology, this design choice is inevitable: If the server system were responsible for the detailed simulation, it would need to render individual video streams for thousands of participants and transmit each of them over the Internet; thus, the server system would require excessive bandwidth and CPU time.

Attacks against semantic integrity exploit the security gap which is inherent in this architecture: they modify the client software as to support behavior which

violates the semantic integrity of the system, but is consistent with the abstract view of the central state server. Since the state server does not have enough information and computation power to verify the decisions of the clients, it is difficult to detect attacks against semantic integrity.

In a previous short paper [Jha et al. 2007] we initiated a systematic study of NVE security. Our analysis identified semantic integrity attacks as the primary new security challenge arising from NVEs. We introduced a protocol, called the Secure Semantic Integrity Protocol (SSIP), which enables the simulation provider to audit the client computations. In the current paper, we analyze the SSIP protocol in depth. First, we show that under common cryptographic assumptions, SSIP detects attacks against semantic integrity with high probability. Since the client software is operating exclusively in the context of NVEs, our formulation of the security property involves a scenario generator which is tailored for the client software and provides a suitable simulation environment. Second, we analyze the bandwidth requirements of the SSIP protocol. We argue that the protocol incurs a very reasonable network overhead, and use an economic model to determine the optimum tradeoff between cost of bandwidth and audit frequency.

Organization. In Section 2 we establish a taxonomy for the security threats arising in the domain of NVEs. In particular, we identify semantic integrity violations as most critical NVE-specific security threat to be handled by the protocol introduced subsequently. In Section 3 we review the client-server protocol, called the client cycle, which is currently employed by most NVEs. Based on the client cycle, we introduce in Section 4 our Secure Semantic Integrity Protocol (SSIP). Then in Section 5, we define the security properties relevant for NVEs which we prove to be satisfied by the SSIP. Finally, in Section 6, we analyze the bandwidth overhead of the SSIP in order to determine a cost-optimal protocol parameterization.

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 [Anderson 2001; Stallings 2003].

(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 [Pritchard 2000], 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 [Debar et al. 1999].

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, called the *client cycle,* that is commonly implemented in NVEs in order to maintain a central abstract state. We write ASTATE to denote this abstracted state, which is centrally maintained by a StateServer. 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 ASTATE[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. On the other hand, 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 [Cousot and Cousot 1977], i.e., $S \in \gamma(\alpha(S))$ and $s = \alpha(S)$ for any $S \in \gamma(s)$.

When client_i connects to the NVE, it first receives a suitable concrete state $S \in \gamma(\text{ASTATE}[\mathsf{client}_i])$ to initialize its local state STATE[client_i]. From this point on, client_i maintains and updates STATE[client_i] locally and communicates only in terms of abstract updates.

If client_i wishes to change its state, it has to inform the 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 STATE[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 $S' = S + \Delta$. 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 (for exemplary numbers from an entertainment application, see Section 6).

We 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 $S' = S + \Delta$ holds, then we require that $\alpha(S') = \alpha(S) + \alpha(\Delta)$ 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 $S' = S + \Delta$, then $\Delta \in \gamma(S, \alpha(\Delta))$ and for all $\Delta' \in \gamma(S, \alpha(\Delta))$, we get $\alpha(S + \Delta') = \alpha(S')$.

Once the client has determined the diff Δ, the client cycle starts and the client sends an abstraction of Δ, called the *request diff* $\delta = \alpha(\Delta)$, to StateServer. Upon

receiving δ, StateServer produces an *authoritative diff* δ' as answer: First, invalid changes requested in δ are removed, and second, changes caused by *other* clients are added to obtain δ'. 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 $\text{STATE}[\mathsf{client}_i]_{t+1} = \text{STATE}[\mathsf{client}_i]_t + \Delta'$. Now a new iteration of the client cycle is being started by computing a new diff Δ.

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 discussed in Section 1: StateServer is only able to check whether the *abstract* request diffs $\delta = \alpha(\Delta)$ are consistent with the *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. Thus, it gains an unfair advantage by ignoring all rules referring to aspects which are abstracted away by $\alpha()$ and which are consequently uncheckable at the server side.

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 the concept of *optimistic execution*: the protocol allows clients to perform any state update which is consistent with the abstract NVE state. At a later time, a dedicated and *fully trusted* AuditServer stochastically performs audits of clients to check whether their sequence of concrete states is indeed consistent with the rules of the NVE. Thus, our approach allows to trade the invested bandwidth for the achieved degree of security.

To facilitate the auditing procedure, 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 client commits to a concrete state; these states will serve as possible starting states for the audit process. Recall that our security model assumes that AuditServer is fully trusted, which implies that a client is unable to alter past action commitments once they are sent. 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.

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. 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.

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 concrete states and $3l$ diffs. In particular, the client has to retain a copy of the concrete 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 $t_a = \lfloor t_0/l - 2 \rfloor l$. 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.

Audit Process. During the auditing, a client 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, the client sends the state information of the current sliding window together with all corresponding StateServer messages to AuditServer. Now, AuditServer checks whether (1) the received state information matches the previously submitted action commitments, (2) whether the client computation is compliant to the rules of the NVE, and (3) 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 on the first concrete full state of each new audit cycle must only arrive upon completion of the corresponding audit cycle.

Protocol Overview. SSIP enforces semantic integrity of clients through 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 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 $M = \text{AuthMsg}(k, m, \mathsf{client})$ as an abbreviation for $m \parallel \text{SignMac}(k, m \parallel \mathsf{client})$, where $\parallel$ denotes a string concatenation with a unique encoding, i.e., $a \parallel b = a' \parallel b'$

implies $a = a'$ and $b = b'$. 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.

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

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

Fig. 2. Protocol ***StatusUpdate***

In this paper, we assume for the sake of simplicity that the audit cycle length l and the pair $\alpha()/\gamma()$ of abstraction and concretization functions are system-wide announced and agreed on parameters. But note that our presentation is always referring to the servers and a *single* client, and therefore, these parameters can be *chosen specifically for each individual client.*

Protocol ***Initialize***. This protocol initializes the state of a client joining the NVE (see Figure 1). 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 $S \in \gamma(\text{ASTATE}[\mathsf{client}])$ to the client. The client initializes its local state S_0 with S. This is the only time throughout the ***Initialize*** and ***StatusUpdate*** protocol when a concrete state is transmitted. Finally, the

(1) AuditServer $\rightsquigarrow$ client : $audit \,\|\, t_0$
(2) client computes $t_a = \left\lfloor \frac{t_0}{l} - 2 \right\rfloor l$
(3) client $\rightsquigarrow$ 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) $\mathrm{VerifyMac}(k, M_i^{(1)} \,\|\, \mathsf{client}, M_i^{(2)}) = \mathrm{true}$ and
  (b) $\mathrm{CFHash}_h(\Delta'_i) = D_i$
(7) AuditServer checks whether $\mathrm{CFHash}_h(S_{t_a}) = Q_{t_a}$ and $\mathrm{CFHash}_h(\hat{S}_{t_a+l}) = Q_{t_a+l}$;
  If $t_a = 0$, client $\rightsquigarrow$ AuditServer : $M_0^{(2)}$ and AuditServer checks whether
  $\mathrm{VerifyMac}(k, S_0 \| \mathsf{client}, M_0^{(2)}) = \mathrm{true}$
(8) AuditServer accepts the computations of client if and only if all tests in steps 5 to 7 passed

Fig. 3. Protocol ***Audit***

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 l-th client cycle is initiated.

Protocol ***StatusUpdate****.* After initialization, the client uses this protocol to update its local state in each client cycle (see Figure 2). 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 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 $M_{t+1} := \mathrm{AuthMsg}(k, \delta'_{t+1} \,\|\, n_t + 1, \mathsf{client})$, 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 $D_{t+1} := \mathrm{CFHash}_h(\Delta'_{t+1})$ 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 $Q_t := \mathrm{CFHash}_h(S_t)$ of its concrete 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. Finally, all outdated audit information is removed.

Protocol ***Audit****.* During the audit protocol, AuditServer validates the computation of one client (see Figure 3). 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 client 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 ($t_a + 1 \leq i \leq t_0$) are unmodified and whether all action commitment messages (D_t and Q_t) submitted by the client beforehand are valid. If the first audit cycle is audited ($t_a = 0$), then client is required to present $M_0^{(2)} = \text{SignMac}(k, S_0 \| \text{client})$ 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.

Protocol 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 concrete state of a client can be processed in background during the l client cycles of an audit cycle. Thus, state updates require only a small amount of additional computation at the client and the server.

The ***Audit*** protocol is more data intensive as it involves the re-simulation of the *semantically relevant* client computations of the *audited fraction* of the client cycles—however, on these grounds, the AuditServer is able to skip large parts of the client computations:

—The audio-visual simulation output and the background scenery simulated for the sake of orientation and immersion only, are semantically irrelevant. Therefore, the AuditServer is free to drop all these computations.

—As the bandwidth analysis in Section 6 suggests, the AuditServer will typically audit a small fraction of client cycles. In our exemplary setting, the AuditServer audits only approximately 15% of the client cycles.

Therefore, in such a setting, we would expect the data center to redo approximately 10% of the client computations. To further reduce the load, we can increase the penalty for cheating in order to deter a larger fraction of potentially malicious participants. Such an approach is expressed in larger values of deterrence factor D in our bandwidth analysis in Section 6. As another strategy, we can audit all clients being located within the same region simultaneously: By checking all movements in a region, we expect to share a large fraction of the necessary computations during re-simulation.

Moreover, the auditing process does not need to be performed on a single machine; rather, several audit servers may be present that audit several clients in parallel—as the protocol suite scales with the number of audit servers without any penalty. Finally, audits are usually not time critical and are relieved from any realtime requirements.

5. SECURITY

In this section, we state the security property achieved by the Secure Semantic Integrity Protocol SSIP. In particular, we introduce the two properties *rule compliance* and *monotone history,* which jointly assure the semantic integrity of the NVE. Then we show that the SSIP enforces both properties during audited client cycles.

We introduce the *successor relation* $\succ$ on (partial) concrete states, where $\text{STATE} \succ \text{STATE}'$ holds if there is a diff Δ such that $\text{STATE}' = \text{STATE} + \Delta$ holds and such that Δ is compliant to the rules of the NVE. We extend the successor relation to abstract

states: If $\text{STATE} \succ \text{STATE}'$ holds, then we require $\alpha(\text{STATE}) \succ \alpha(\text{STATE}')$ to hold. Since we do not require the converse, the successor relation on abstract states is allowed to *overapproximate* the permitted concrete behavior. Using the successor relation, we say that a sequence $\langle \text{STATE}_0, \ldots, \text{STATE}_t \rangle$ of concrete states is *valid* if $\text{STATE}_i \succ \text{STATE}_{i+1}$ holds for all $0 \leq i < t$. Analogously, $\langle \text{ASTATE}_0, \ldots, \text{ASTATE}_t \rangle$ is a *valid* sequence of abstract states if $\text{ASTATE}_i \succ \text{ASTATE}_{i+1}$ holds for all $0 \leq i < t$.

In the following, we use A_t as abbreviation for $\text{ASTATE}[\mathsf{client}]_t$, similarly as we use S_t as abbreviation for $\text{STATE}[\mathsf{client}]$. In the course of the ***Initialize***-protocol and t subsequent iterations of the ***StatusUpdate***-protocol, the client receives a concrete state S_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 $\langle A_0, \ldots, A_t \rangle$, such that $A_0 = \alpha(S_0)$ and $A_{i+1} = A_i + \delta'_{i+1}$. The StateServer is only able to check whether the abstract sequence $\langle A_0, \ldots, A_t \rangle$ is valid. But the validity of the abstract sequence does not guarantee its *realizability:* We say that $\langle A_0, \ldots, A_t \rangle$ is *realizable at* concrete state S_0, if there exists a valid concrete sequence $\langle S_0, \ldots, S_t \rangle$ which starts with S_0 and where $S_i \in \gamma(A_i)$ holds for all $0 \leq i \leq t$.

Definition 5.1 *Rule Compliance.* A client behaves *rule compliant,* iff the sequence of abstract states $\langle A_0, \ldots, A_t \rangle$ produced by the client and the StateServer is realizable at the concrete state S_0 received by the client during the ***Initialize***-protocol.

If the ***Audit***-protocol is initiated at round t_0, the AuditServer asks the client to disclose the sequence states $\langle S_{t_a}, \ldots, S_{t_0} \rangle$ of concrete states by transmitting S_{t_a} and the diffs $\Delta_{t_a+1}, \ldots, \Delta_{t_0}$. In this situation, a malicious client could provide a rule compliant sequence which has been manipulated: Using the knowledge available at t_0, the client could rewrite its own history to gain some advantage.

For a given $q \geq t$, we denote with H_t^q the state returned by the client if queried at time q for its former state S_t. H_t^q is called a *historical state.* If the client is honest, it will always return its truthful historical states, i.e., $H_t^q = S_t$ for all $q \geq t$. In this case, the client never rewrites its history (i.e., there exists no pair $H_t^q \neq H_t^r$) and thus we say that the client has a monotone history. Finally, a monotone history together with rule compliance assures honest behavior.

Definition 5.2 *Monotone History.* A client has a monotone history, iff $H_t^q = S_t$ for all $q \geq t$.

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

Using standard cryptographic assumptions, the Secure Semantic Integrity Protocol (SSIP) enforces honest client behavior:

Claim 5.4 Security of 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).*

In order to refine and formally prove the statement of Claim 5.4, we show that whenever a client is able to behave maliciously within a reasonable demonstration

environment 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. Note that by assumption AuditServer is fully trusted and the client can therefore only tamper with data sent to the state or audit servers.

We 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 ScenarioGenerator in our proof which provides a client with a realistic environment by pretending to be the StateServer. Thus, the client interacts with the ScenarioGenerator in the same way as it would interact with the StateServer.

Definition 5.5 *Scenario Generator.* ScenarioGenerator is a probabilistic interactive Turing Machine which takes the security parameter 1^n as initial input and produces interactively with a client some scenario $\langle S_0, \delta'_1, \ldots, \delta'_m \rangle$ where m is polynomial in n. The resulting sequence $\langle A_0, \ldots, A_t \rangle$ of abstract states with $A_0 = \alpha(S_0)$ and $A_{i+1} = A_i + \delta'_{i+1}$ must be valid. After computing and outputting the first concrete state S_0, the ScenarioGenerator waits for an abstract request diff δ_t from the client to compute and return a corresponding authoritative diff δ'_t. Once m authoritative diffs have been produced, the ScenarioGenerator terminates. The computation of the initial concrete state S_0 and the computation of each abstract diff δ_t must respect probabilistic polynomial time bounds with respect to n.

Each malicious client is assumed to come with a suitable ScenarioGenerator assisting the client in cheating successfully: It produces a scenario which allows the corresponding client to demonstrate its ability to behave maliciously. For example, the client could require the simultaneous presence of two specific objects in order to perform its exploit and such a situation might arise infrequently in real simulations. In such a case, the ScenarioGenerator will provide the client with a matching scenario to apply its exploit. While the ScenarioGenerator is designed to provide an opportunity for a client to cheat, it is at the same time bound to respect the abstract rules of the NVE since $\langle A_0, \ldots, A_t \rangle$ must be a valid sequence. However, strictly speaking, it is not necessary to explicitly provide a suitable ScenarioGenerator for a given malicious client, it is only required that such a ScenarioGenerator exists.

The goal of the client in interacting with the ScenarioGenerator is to behave maliciously while being undetected by the AuditServer. Thus, if the client is successful in doing so, it demonstrated its capability to cheat under certain realistic circumstances. More precisely, to prove the malicious capabilities of a client, the allegedly malicious client, its associated ScenarioGenerator, and the trusted real-life AuditServer engage in the following experiment.

Definition 5.6 *Malicious Behavior Experiment.* In the malicious behavior experiment, a client, a ScenarioGenerator, 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 client and ScenarioGenerator 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 current local 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 by the AuditServer.

Fig. 4 depicts this experiment graphically: GenMac generates a random MAC-key which is subsequently used by AuthMsg to authorize the messages originating from the ScenarioGenerator. The ScenarioGenerator produces the initial state S_0 and the authoritative updates $\delta'_1, \ldots, \delta'_m$, and authenticates them with the help of AuthMsg. The authenticated messages $M_0, \ldots, M_m$ are sent to client and forwarded (possibly modified) to AuditServer as in the real ***StatusUpdate***-protocol.

AuditServer also receives the hashes $D_1, \ldots, D_m$ on the concrete diffs and the hashes $Q_0, \ldots, Q_{\lfloor m/l \rfloor l}$ on the complete concrete states. Finally, the client produces the sequence of local states $S_0, \ldots, S_m$. In the figure, we use thin lines to depict messages sent during the ***Initialize***- or ***StatusUpdate***-protocol.

In contrast, 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, the 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 Claim 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 secure 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:

THEOREM 5.7 SECURITY OF SSIP. *If CFHash is a collection of collision-free hash functions and SignMac a Message Authentication Code secure against selective forgery of messages, then the Secure Semantic Integrity Protocol (SSIP) guarantees that any malicious behavior experiment with a probabilistic polynomial-time client has negligible success probability.*

To prove Theorem 5.7, we show that a client which passes the malicious behavior experiment with a non-negligible probability will yield a procedure AttackHash producing collisions for the allegedly collision-free hash function and a procedure AttackMac which forges MACs for the allegedly secure MAC function. *At least one of these two procedures* will be successful with non-negligible probability, assuming that the malicious behavior experiment succeeds 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. Thus SSIP enforces honest behavior during audited client cycles.

PROOF OF THEOREM 5.7. Suppose there exist a client and a ScenarioGenerator such that the client succeeds in the malicious behavior experiment with non-

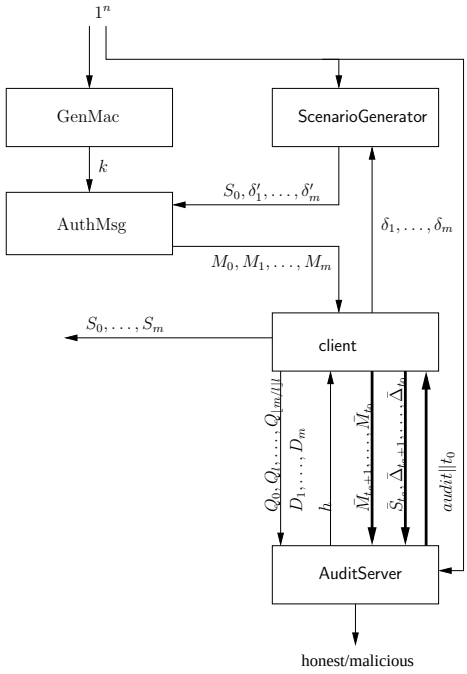


Fig. 4. Malicious Behavior Experiment

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 5.7.

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:

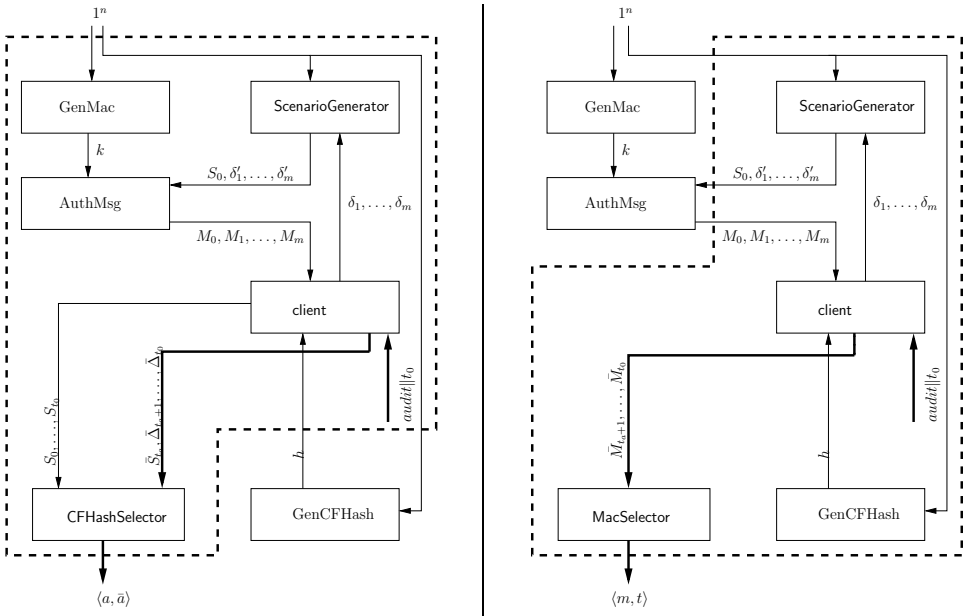
(1) The client cheats on the first audited state S_{t_a}, i.e., the state $\bar{S}_{t_a}$ sent to AuditServer differs from S_{t_a}. In this case the client has found a second pre-image of the hash function, as $\text{CFHash}_h(S_{t_a}) = \text{CFHash}_h(\bar{S}_{t_a})$.

(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 provides a message $\bar{M}_i = \text{AuthMsg}(k, \bar{\delta}'_i \parallel n_i, \mathsf{client})$ 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 $\bar{\delta}'_i \parallel n_i \parallel \mathsf{client}$, 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 leaves 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 pre-image 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 $\bar{S}_{t_a+l} = \bar{S}_{t_a} + \bar{\Delta}_{t_a+1} + \ldots + \bar{\Delta}_{t_a+l}$. 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 of 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 Fig. 5a: The attack procedure AttackHash receives the index h of the hash 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 the message $audit \| t_0$, which starts the ***Audit***-protocol, is sent at a uniformly and randomly chosen point in time.
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 $\langle S_{t_a}, \bar{S}_{t_a} \rangle$ and $\langle S_{t_a+l}, \bar{S}_{t_a+l} \rangle$ and from the sequence of pairs $\langle \Delta'_{t_a+1}, \bar{\Delta}'_{t_a+1} \rangle, \ldots, \langle \Delta'_{t_0}, \bar{\Delta}'_{t_0} \rangle$.



(a) AttackHash (b) AttackMac

Fig. 5. Attack Procedures

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 concrete 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 $t = \text{SignMac}(k, m)$ holds for a key k which is not known to the AttackMac.

In Fig. 5b, 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 concrete 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. □

Integrity of the Audit Log. In the construction of SSIP we assumed that both StateServer and AuditServer are fully trusted; in particular, AuditServer does not frame a client by altering its previously sent action commitments. To prevent tampering of the audit log by a potentially malicious AuditServer, the client can protect the auditing data by using a cryptographic signature instead of a hash function. By replacing CFHash_h with a signing algorithm of an cryptographic signature scheme which secure against existential forgery under a chosen message attack, the integrity and authenticity of the audit information can be assured. The security of this modified SSIP protocol can be proven completely analogous to Theorem 5.7 by a simultaneous reduction against the unforgeability of the MAC and the signature scheme in use.

6. BANDWIDTH OVERHEAD

In the previous section we showed that SSIP reliably identifies malicious behavior in audited client cycles. However, due to the computational and networking resources required for auditing, it is not possible to audit each cycle of all clients participating in an NVE. Thus, a stochastic approach must be applied, which selectively audits a fraction of the client cycles. It remains to determine the optimal audit probability which balances the cost of additional bandwidth on the one hand, and the cost of successful and uncaught cheat attempts on the other hand. Hence, we first establish a lower bound on the minimal probability that a client cycle is audited as function of the bandwidth, and second we calculate the optimal bandwidth investment in a reasonable cost model.

Relating Audit Initiation Probability and Bandwidth. As a model for the auditing strategy, we assume that AuditServer is deciding randomly and uniformly with the *audit initiation probability* p_{audit} whether to initiate the ***Audit***-protocol within

a client cycle.[1] To relate p_{audit} with the resulting bandwidth requirements, we use the following notation: $|S|$, $|\Delta|$, and $|\delta|$ denote the *average* size of the client-side maintained concrete state, of a concrete diff between two subsequent client states, and of an abstract diff between two subsequent client states, respectively. Furthermore, h and m denote the size of a hash and a MAC-tag.

Starting with the *downstream bandwidth,* we note that there are no additional messages sent from the servers to the client. Only each authoritative diff δ' is authorized with a MAC-tag. However, since these MAC-tags are fairly small and must be added to the authoritative diffs irrespective of all other protocol parameters, we ignore them subsequently and rather discuss the upstream bandwidth requirements.

The *upstream bandwidth* U involves the messages of the ***Audit***-protocol as well as the additional messages of the ***StatusUpdate***-protocol. During the ***StatusUpdate***-protocol we have to account for the request diff, a hash of the concrete diff of the local state, and a hash of the complete local state in each lth client cycle, yielding $|\delta| + h\,(1 + 1/l)$ bytes sent on average from the client to the servers in a single client cycle of ***StatusUpdate***. If an ***Audit***-protocol is issued, the client must transmit a copy of its complete local state and between $2l + 1$ and $3l$ concrete diffs. Each of these concrete diffs must be accompanied by their corresponding authoritative diffs and their MAC-tags. Taken together, each time a client is audited, it has to transmit on average $|S| + 3l(|\Delta| + |\delta| + m)$ bytes.

Since we assume that an ***Audit***-protocol is issued with probability p_{audit} during a client cycle, the expected bandwidth requirements U for upstream messages are bounded by $U \leq |\delta| + h\,(1 + 1/l) + p_{audit}\,(|S| + 3l(|\Delta| + |\delta| + m))$, and therefore we obtain

$$\frac{U - |\delta| - h\,(1 + 1/l)}{|S| + 3l(|\Delta| + |\delta| + m)} \leq p_{audit}\ .$$

Note that any sensible choice for U must be larger than $|\delta| + h$ as to cover the abstract request diff δ and the hashes. On the other hand, choosing U larger than $|\Delta|$ does not make sense at all since then the client could send concrete diffs instead of abstract diffs—thus rendering the SSIP unnecessary.[2] Therefore we have

$$|\delta| + h < U < |\Delta|$$

as suitable range for U.

Relating Minimal Audit Probability and Bandwidth. To analyze the probability that a malicious client is caught by the AuditServer, we introduce the *minimal audit probability* $p_{\min}$ as the probability that a single client cycle is audited. The particular client cycle performing the transition from S_t to S_{t+1} is audited whenever the ***Audit***-protocol is initiated at some round t_0 with $t_a \leq t < t_0$ for $t_a = \lfloor t_0/l - 2 \rfloor\, l$. This is the case for $3l - (t - \lfloor t/l \rfloor l) - 1$ individual values of t_0. Thus the minimal probability that this particular cycle is audited is given by

[1] In a practical implementation, the AuditServer will also consider the client behavior, e.g., AuditServer will preferably audit clients which show a suspiciously strong performance.

[2] Even if the clients send concrete diffs, the server system could still use abstraction to reduce its CPU load. In this case, the server system could audit its clients independently from the clients.

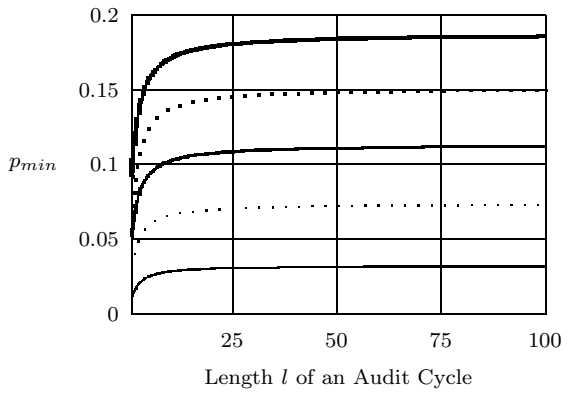


Fig. 6. Lower Bound on $p_{\min}$

$1-(1-p_{audit})^{3l-(t-\lfloor t/l \rfloor l)-1} \geq 1-(1-p_{audit})^{2l} = p_{\min}$. Therefore, we find

$$\frac{U-|\delta|-h\,(1+1/l)}{|S|+3l(|\Delta|+|\delta|+m)} \leq p_{audit} = 1-(1-p_{\min})^{1/2l}$$

and obtain the following lemma:

LEMMA 6.1 LOWER BOUND ON $p_{\min}$. *For an average upstream bandwidth U and an audit cycle length l, the minimal audit probability $p_{\min}$ that an individual client cycle is audited is bounded by*

$$1-\left[1-\frac{U-|\delta|-h\,(1+1/l)}{|S|+3l(|\Delta|+|\delta|+m)}\right]^{2l} \leq p_{\min}\,.$$

The values of all variables which occur in Lemma 6.1—with the exception of U, l and $p_{\min}$—are determined by the underlying NVE and the concrete choice of the algorithms and parameters for computing MAC-tags and collision-free hashes. Thus, the lemma presents a trade-off between U and $p_{\min}$ where l can be chosen freely to optimize the result: The larger the bandwidth U, the larger becomes $p_{\min}$—where larger values of l increase the probability $p_{\min}$ without requiring additional bandwidth.

In Fig. 6, we show the resulting lower bound for $p_{\min}$ as function of l where we set U to 6, 7, 8, 9, and $10kB$, respectively (the higher the bandwidth U, the higher $p_{\min}$). The remaining variables are fixed as follows: $|S| = 50kB$ (corresponding to 500 entities in the ares of interest of a client with each having roughly 100 bytes of dynamic content), $|\Delta| = 10kB$, $|\delta| = 5kB$, $h = 256B$, and $m = 256B$.[3] For these values, the suitable range for U is given by $5376B = |\delta| + h < U < |\Delta| = 10240B$.

For fixed U, $p_{\min}$ increases and quickly converges as l goes to infinity. Thus, we can improve the minimal auditing probability $p_{\min}$ by increasing l, but as the bound approaches its limit, further increments of l only achieve a minimal effect.

[3]These numbers and all subsequently mentioned figures rely on industrial experience within gaming development.

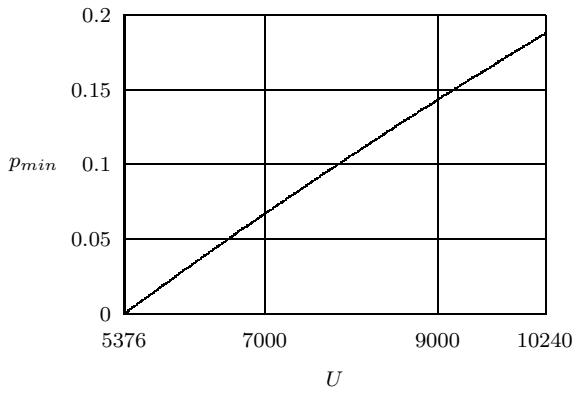


Fig. 7. Minimal Audit Probability $p_{\min}$

Therefore, we use the bound of Lemma 6.1 with l going to infinity to obtain a simpler approximate lower bound for $p_{\min}$, shown in the next lemma (for a detailed proof, see Appendix A).

LEMMA 6.2 LIMIT ON THE LOWER BOUND FOR $p_{\min}$. *As the length l of an audit cycle goes to infinity, the lower bound on $p_{\min}$ converges to*

$$1 - e^{-\frac{2(U-|\delta|-h)}{3(|\Delta|+|\delta|+m)}} \leq p_{\min}.$$

Fig. 7 shows the resulting relationship where we use for all parameters the same values as for Fig. 6. Given these values, the bandwidth U must be chosen from the interval $5376 < U < 10240$. For these possibles choices, the SSIP guarantees a minimal audit probability $p_{\min}$ strictly between 0 (for U being close to 5376) and 0.185 (for U being almost 10240).

Optimizing the Protocol Parameters. To optimize the invested bandwidth, we have to establish a cost model which relates the bandwidth requirements with the financial loss incurred due to successful cheaters. To do so, we introduce the *bandwidth cost* C_B per transmitted byte and the *cheating cost* C_C for each successfully and maliciously executed client cycle. We relate these costs with each other by computing the *total protocol cost* C_T which describes the combined bandwidth and cheating cost for a single average client cycle. Next we introduce the *success probability* p_S for a client to perform an undetected malicious client cycle such that we have $C_T = UC_B + p_S C_C$ for the total cost.

This success probability p_S depends on a number of parameters. In particular, we have to consider the deterring effect of caught cheaters: If many cheaters are caught and punished, the fraction of cheating participants will decline, i.e., potential cheaters will be deterred. Thus, we subdivide the *potentially malicious fraction* p_M into (1) the uncaught fraction p_S of successful cheaters, (2) the *caught fraction* p_C of cheaters, and (3) the *deterred fraction* p_D, with $p_M = p_S + p_C + p_D$. Note that $p_M - p_D$ is then the fraction of players who attempt to cheat. Following approaches established in economics [Levitt 1996], we set $p_D = Dp_C$ for a *deterrence factor*

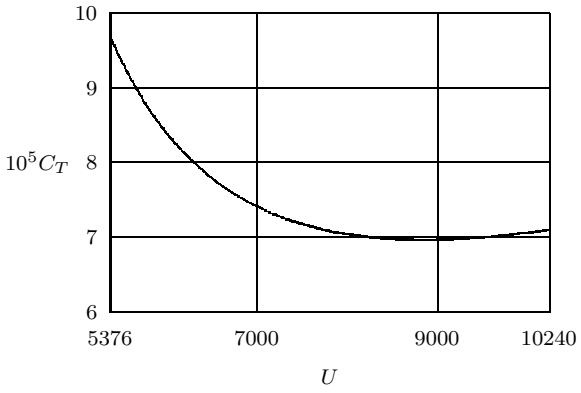


Fig. 8. Total Protocol Cost C_T

D. Therefore, as a client cycle is audited with probability $p_{\min}$, we get $p_C = p_{\min}(p_M - p_D) = p_{\min}(p_M - Dp_C)$ and arrive at $p_C = p_{\min}p_M/(1 + Dp_{\min})$. Since $p_S = p_M - p_C - p_D = p_M - (1+D)p_C$ we obtain—using $\alpha = \frac{3}{2}(|\Delta|+|\delta|+m)$ and $\beta = 1 - p_{\min} = e^{-(U-|\delta|-h)/\alpha}$ as abbreviations—the equation $p_S = p_M\beta/(1 + D - D\beta)$. Thus, the total protocol cost C_T is given by

$$C_T = UC_B + p_M C_C \frac{\beta}{1 + D - D\beta}, \tag{1}$$

which we want to minimize. In Fig. 8, we plot C_T as function in U where we use $C_B = 5 \cdot 10^{-9}$, $C_C = 10^{-4}$, $D = 10$, and $p_M = 7/10$ in addition to the parameters already fixed for the preceding figures. The optimal choice for U in this example is $U \approx 8855.86$ which can be calculated directly by equating the derivate of Eq. 1 with 0, and solving the resulting theorem (for a detailed proof, see Appendix A):

THEOREM 6.3 OPTIMAL CHOICE FOR THE BANDWIDTH U. *The optimal choice for the bandwidth U in the above cost model is*

$$U = |\delta| + h - \alpha \log\left[1 + \frac{1}{D} + \frac{1}{2D^2\gamma}\left(1 - \sqrt{4D^2\gamma + 4D\gamma + 1}\right)\right],$$

where we use $\alpha = \frac{3}{2}(|\Delta| + |\delta| + m)$ and $\gamma = \frac{C_B\alpha}{(1+D)p_M C_C}$ as abbreviations.

7. CONCLUSIONS

The Secure Semantic Integrity Protocol (SSIP) bridges the semantic gap arising as an inherent consequence of abstraction-based client-server NVE architectures. In this paper, we introduced the *malicious behavior experiment* to analyze the SSIP in depth and to prove that the audit trail mechanism of the SSIP is *provably secure.* Thus, an audited cheating client has only a negligible chance of not being caught by the AuditServer. However, due to bandwidth restrictions, auditing is only applicable in a fraction of the occurring client cycles. Hence, a trade-off between the audit probability and the resulting bandwidth cost must be chosen. We introduced a

suitable cost model and determined a correspondingly *optimal choice for the audit probability*.

Acknowledgements.. We thank Hovav Shacham for his helpful comments on a preliminary version of this paper.

REFERENCES

ACM 2003. A game experience in every application (Special Issue). *Communications of the ACM 46,* 7 (July).

ACM 2004. Interactive Immersion in 3D Graphics (Special Issue). *Communications of the ACM 47,* 8 (August).

Anderson, R. 2001. *Security Engineering*. Wiley.

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

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

Economist 2007a. A credit crunch in cyberspace. Trouble in paradise. *The Economist*. `http://www.economist.com/finance/displaystory.cfm?story_id=9661900`.

Economist 2007b. Online gaming's Netscape moment? The Economist. `http://www.economist.com/search/displaystory.cfm?story_id=9249157`.

Gardner, J. 2007. Virtual world of 'Second Life' is starting to look a lot like Sweden. *San Francisco Business Times*.

Heuser, H. 1998. *Lehrbuch der Analysis, Teil 1*, 12. ed. Teubner.

IEEE 2004. Networked Virtual Environments (Special Issue). *IEEE Commmunications 42,* 4 (April).

Jha, S., Katzenbeisser, S., Schallhart, C., Veith, H., and Chenny, S. 2007. Enforcing Semantic Integrity on Untrusted Clients in Networked Virtual Environments (Extended Abstract). In *IEEE Security and Privacy*. 179–186.

Levitt, S. 1996. The Effect of Prison Population on Crime Rates: Evidence from Prison Overcrowding Litigation. *The Quaterly Journal of Economics 111,* 2, 319–351.

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

Siklos, R. 2006. A Virtual World but Real Money. *New York Times*. `http://www.nytimes.com/2006/10/19/technology/19virtual.htm`.

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

A. PROOFS FOR SECTION 6

Proof of Lemma 6.2. One has to show that the left-hand side of Lemma 6.1, i.e.,

$$1 - \left[1 - \frac{U - |\delta| - h\,(1 + 1/l)}{|S| + 3l(|\Delta| + |\delta| + m)}\right]^{2l},$$

converges to the lower bound Lemma 6.2, i.e.,

$$1 - e^{-\frac{2(U-|\delta|-h)}{3(|\Delta|+|\delta|+m)}}$$

as l goes to infinity. To this end, we use the identity [Heuser 1998]

$$\lim_{n\to\infty}\left(1 + \frac{a_n}{b + cn}\right)^n = e^{a/c}$$

for $\lim_{n\to\infty} a_n = a$ and introduce abbreviations $a_l = U - |\delta| - h\,(1 + 1/l)$, $b = |S|$, and $c = 3(|\Delta| + |\delta| + m)$ in order to rewrite the required limes

$$\lim_{l\to\infty}\left(1 - \left[1 - \frac{U - |\delta| - h\,(1+1/l)}{|S| + 3l(|\Delta| + |\delta| + m)}\right]^{2l}\right)$$

as

$$\lim_{l\to\infty} 1 - \left[1 - \frac{a_l}{b + cl}\right]^{2l}.$$

Then we obtain the statement of the Lemma as follows:

$$\begin{aligned}\lim_{l\to\infty} 1 - \left[1 - \frac{a_l}{b+cl}\right]^{2l} &= 1 - \left(\lim_{l\to\infty}\left[1 - \frac{a_l}{b+cl}\right]^{l}\right)^2 \\ &= 1 - e^{-2a/c} \\ &= 1 - e^{-\frac{2(U-|\delta|-h)}{3(|\Delta|+|\delta|+m)}}\end{aligned}$$

□

Proof of Theorem 6.3. Using

$$\alpha = \frac{3}{2}(|\Delta| + |\delta| + m)\ ,\ \beta = e^{-(U-|\delta|-h)/\alpha}\ ,\ \text{and}\ \gamma = \frac{C_B\alpha}{(1+D)p_M C_C}$$

as abbreviations, we have to prove that

$$C_T = UC_B + p_M C_C \frac{\beta}{1 + D - D\beta}$$

becomes minimal for

$$U = |\delta| + h - \alpha \log\left[1 + \frac{1}{D} + \frac{1}{2D^2\gamma}\left(1 - \sqrt{4D^2\gamma + 4D\gamma + 1}\right)\right]\ .$$

Calculating the derivate of C_T in U, we obtain

$$C_T' = C_B - p_M C_C \frac{(1+D)\beta}{\alpha(1 + D - D\beta)^2}$$

and by equating this derivate with 0 and solving the resulting equation for U we find the two solutions

$$U = |\delta| + h - \alpha \log\left[1 + \frac{1}{D} + \frac{1}{2D^2\gamma}\left(1 \pm \sqrt{4D^2\gamma + 4D\gamma + 1}\right)\right] \tag{2}$$

for the desired extremal point. As the second derivate

$$C_T'' = p_M C_C \frac{(1+D)((1+D)^2\beta - D^2\beta^3)}{\alpha^2(1 + D - D\beta)^4}$$

becomes positive if and only if we choose in Eq. 2 to subtract the square root, we obtain the desired minimum for C_T. □

XXX — RECEIVED BLOCK