Accountable Virtual Machines

Andreas Haeberlen Paarijaat Aditya Rodrigo Rodrigues Peter Druschel
University of Pennsylvania Max Planck Institute for Software Systems (MPI-SWS)

MPI-SWS Technical Report 2010-003*

Abstract

In this paper, we introduce *accountable virtual machines (AVMs)*. Like ordinary virtual machines, AVMs can execute binary software images in a virtualized copy of a computer system; in addition, they can record non-repudiable information that allows auditors to subsequently check whether the software behaved as intended. AVMs provide strong accountability, which is important, for instance, in distributed systems where different hosts and organizations do not necessarily trust each other, or where software is hosted on third-party operated platforms. AVMs can provide accountability for unmodified binary images and do not require trusted hardware. To demonstrate that AVMs are practical, we have designed and implemented a prototype AVM monitor based on VMware Workstation, and used it to detect several existing cheats in Counterstrike, a popular online multi-player game.

1 Introduction

An *accountable virtual machine (AVM)* provides users with the capability to audit the execution of a software system by obtaining a log of the execution, and comparing it to a known-good execution. This capability is particularly useful when users rely on software and services running on machines owned or operated by third parties. Auditing works for any binary image that executes inside the AVM and does not require that the user trust either the hardware or the accountable virtual machine monitor on which the image executes. Several classes of systems exemplify scenarios where AVMs are useful:

- in a competitive system, such as an online game or an auction, users may wish to verify that other players do not cheat, and that the provider of the service implements the stated rules faithfully;
- nodes in peer-to-peer and federated systems may wish to verify that others follow the protocol and contribute their fair share of resources;
- cloud computing customers may wish to verify that the provider executes their code as intended.

In these scenarios, software and hardware faults, misconfigurations, break-ins, and deliberate manipulation can lead to an abnormal execution, which can be costly to users and operators, and may be difficult to detect. When such a malfunction occurs, it is difficult to establish who is responsible for the problem, and even more challenging to produce evidence that proves a party's innocence or guilt. For example, in a cloud computing environment, failures can be caused both by bugs in the customer's software and by faults or misconfiguration of the provider's platform. If the failure was the result of a bug, the provider would like to be able to prove his own innocence, and if the provider was at fault, the customer would like to obtain proof of that fact.

AVMs address these problems by providing users with the capability to detect faults, to identify the faulty node, and to produce *evidence* that connects the fault to the machine that caused it. These properties are achieved by running systems inside a virtual machine that 1) maintains a log with enough information to reproduce the entire execution of the system, and that 2) associates each outgoing message with a cryptographic record that links that action to the log of the execution that produced it. The log enables users to detect faults by replaying segments of the execution using a known-good copy of the system, and by cross-checking the externally visible behavior of that copy with the previously observed behavior. AVMs can provide this capability for any black-box binary image that can be run inside a VM.

AVMs detect integrity violations of an execution without requiring the audited machine to run hardware or software components that are trusted by the auditor. When such trusted components are available, AVMs can be extended to detect some confidentiality violations as well, such as private data leaking out of the AVM.

This paper makes three contributions: 1) it introduces the concept of AVMs, 2) it presents the design of an *accountable virtual machine monitor (AVMM)*, and 3) it demonstrates that AVMs are practical for a specific application, namely the detection of cheating in multi-

*Extended version of a paper that appears in the proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation Vancouver, BC, Canada October 4-6, 2010 player games. Cheat detection is an interesting example application because it is a serious and well-understood problem for which AVMs are effective: they can detect a large and general class of cheats. Out of 26 existing cheats we downloaded from the Internet, AVMs can detect every single one—without prior knowledge of the cheat's nature or implementation.

We have built a prototype AVMM based on VMware Workstation, and used it to detect real cheats in Counterstrike, a popular multi-player game. Our evaluation shows that the costs of accountability in this context are moderate: the frame rate drops by 13%, from 158 fps on bare hardware to 137 fps on our prototype, the ping time increases by about 5 ms, and each player must store or transmit a log that grows by about 148 MB per hour after compression. Most of this overhead is caused by logging the execution; the additional cost for accountability is comparatively small. The log can be transferred to other players and replayed there during the game (online) or after the game has finished (offline).

While our evaluation in this paper focuses on games as an example application, AVMs are useful in other contexts, e.g., in p2p and federated systems, or to verify that a cloud platform is providing its services correctly and is allocating the promised resources [18]. Our prototype AVMM already supports techniques such as partial audits that would be useful for such applications, but a full evaluation is beyond the scope of this paper.

The rest of this paper is structured as follows. Section 2 discusses related work, Section 3 explains the AVM approach, and Section 4 presents the design of our prototype AVMM. Sections 5 and 6 describe our implementation and report evaluation results in the context of games. Section 7 describes other applications and possible extensions, and Section 8 concludes this paper.

2 Related work

Deterministic replay: Our prototype AVMM relies on the ability to replay the execution of a virtual machine. Replay techniques have been studied for more than two decades, usually in the context of debugging, and mature solutions are available [6, 15, 16, 39]. However, replay by itself is not sufficient to detect faults on a remote machine, since the machine could record incorrect information in such a way that the replay looks correct, or provide inconsistent information to different auditors.

Improving the efficiency of replay is an active research area. Remus [11] contributes a highly efficient snapshotting mechanism, and many current efforts seek to improve the efficiency of logging and replay for multi-core systems [13, 16, 28, 29]. AVMMs can directly benefit from these innovations.

Accountability: Accountability in distributed systems has been suggested as a means to achieve practical security [26], to create an incentive for cooperative behavior [14], to foster innovation and competition in the Internet [4, 27], and even as a general design goal for dependable networked systems [43]. Several prior systems provide accountability for specific applications, including network storage services [44], peer-to-peer content distribution networks [31], and interdomain routing [2, 19]. Unlike these systems, AVMs are application independent. PeerReview [21] provides accountability for general distributed systems. However, PeerReview must be closely integrated with the application, which requires source code modifications and a detailed understanding of the application logic. It would be impractical to apply PeerReview to an entire VM image with dozens of applications and without access to the source code of each. AVMs do not have these limitations; they can make software accountable 'out of the box'.

Remote fault detection: GridCop [42] is a compiler-based technique that can be used to monitor the progress and execution of a remotely executing program by inspecting periodic beacon packets. GridCop is designed for a less hostile environment than AVMs: it assumes a trusted platform and self-interested hosts. Also, GridCop does not work for unmodified binaries, and it cannot produce evidence that would convince a third party that a fault did or did not happen.

A trusted computing platform can be used to detect if a node is running modified software [17, 30]. The approach requires trusted hardware, a trusted OS kernel, and a software and hardware certification infrastructure. Pioneer [36] can detect such modifications using only software, but it relies on recognizing sub-millisecond delay variations, which restricts its use to small networks. AVMs do not require any trusted hardware and can be used in wide-area networks.

Cheat detection: Cheating in online games is an important problem that affects game players and game operators alike [24]. Several cheat detection techniques are available, such as scanning for known hacks [23, 35] or defenses against specific forms of cheating [7, 32]. In contrast to these, AVMs are generic; that is, they are effective against an entire class of cheats. Chambers et al. [9] describe another technique to detect if players lie about their game state. The system relies on a form of tamper-evident logs; however, the log must be integrated with the game, while AVMs work for unmodified games.

3 Accountable Virtual Machines

3.1 Goals

Figure 1 depicts the basic scenario we are concerned with in this paper. Alice is relying on Bob to run some software S on a machine M, which is under Bob's con-

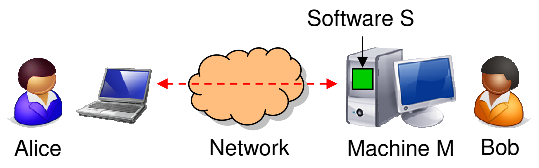


Figure 1: Basic scenario. Alice is relying on software S, which is running on a machine that is under Bob's control. Alice would like to verify that the machine is working properly, and that Bob has not modified S.

trol. However, Alice cannot observe M directly, she can only communicate with it over the network. Our goal is to enable Alice to check whether M behaves as expected, without having to trust Bob, M, or any software running on M.

To define the behavior Alice expects M to have, we assume that Alice has some reference implementation of M called M_R, which runs S. We say that M is correct iff M_R can produce the same network output as M when it is started in the same initial state and given precisely the same network inputs. If M is not correct, we say that it is *faulty*. This can happen if M differs from M_R, or Bob has installed software other than S. Our goal is to provide the following properties:

- **Detection:** If M is faulty, Alice can detect this.
- **Evidence:** When Alice detects a fault on M, she can obtain evidence that would convince a third party that M is faulty, without requiring that this party trust Alice or Bob.

We are particularly interested in solutions that work for any software S that can execute on M and M_R. For example, S could be a program binary that was compiled by someone other than Alice, it could be a complex application whose details neither Alice nor Bob understand, or it could be an entire operating system image running a commodity OS like Linux or Windows.

In the rest of this paper, we will omit explicit references to S when it is clear from the context which software M is expected to run.

3.2 Approach

To detect faults on M, Alice must be able to answer two questions: 1) which exact sequence of network messages did M send and receive, and 2) is there a correct execution of M_R that is consistent with this sequence of messages? Answering the former is not trivial because a faulty M—or a malicious Bob—could try to falsify the answer. Answering the latter is difficult because the number of possible executions for any nontrivial software is large.

Alice can solve this problem by combining two seemingly unrelated technologies: tamper-evident logs and virtual machines. A *tamper-evident log* [21] requires each node to record all the messages it has sent or received. Whenever a message is transmitted, the sender and the receiver must prove to each other that they have added the message to their logs, and they must commit to the contents of their logs by exchanging an *authenticator* – essentially, a signed hash of the log. The authenticators provide nonrepudiation, and they can be used to detect when a node tampers with its log, e.g., by forging, omitting, or modifying messages, or by forking the log.

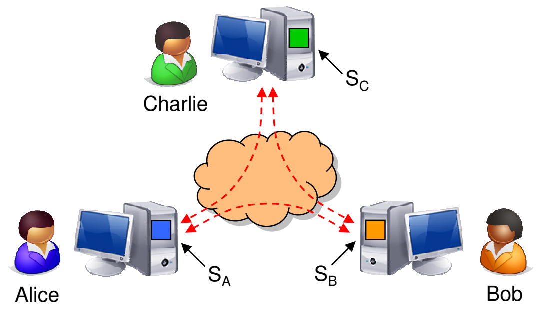
Once Alice has determined that M's message log is genuine, she must either find a correct execution of M_R that matches this log, or establish that there isn't one. To help Alice with this task, M can be required to record additional information about nondeterministic events in the execution of S. Given this information, Alice can use deterministic replay [8, 15] to find the correct execution on M_R, provided that one exists.

Recording the relevant nondeterministic events seems difficult at first because we have assumed that neither Alice nor Bob have the expertise to make modifications to S; however, Bob can avoid this by using a *virtual machine monitor (VMM)* to monitor the execution of S and to capture inputs and nondeterministic events in a generic, application-independent way.

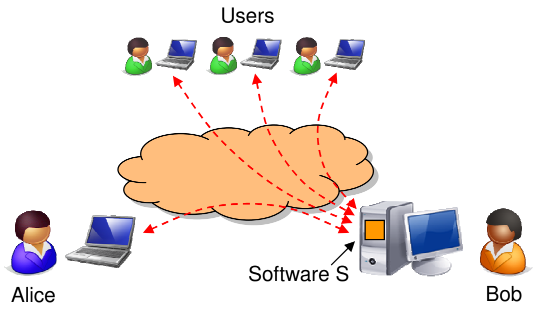
3.3 AVM monitors

The above building blocks can be combined to construct an *accountable virtual machine monitor (AVMM)*, which implements AVMs. Alice and Bob can use an AVMM to achieve the goals from Section 3.1 as follows:

1. Bob installs an AVMM on his computer and runs the software S inside an AVM. (From this point forward, M refers to the entire stack consisting of Bob's computer, the AVMM running on Bob's computer, and Alice's virtual machine image S, which runs on the AVMM.)
2. The AVMM maintains a tamper-evident log of the messages M sends or receives, and it also records any nondeterministic events that affect S.
3. When Alice receives a message from M, she detaches the authenticator and saves it for later.
4. Alice periodically audits M as follows: she asks the AVMM for its log, verifies it against the authenticators she has collected, and then uses deterministic replay to check the log for faults.
5. If replay fails or the log cannot be verified against one of the authenticators, Alice can give M_R, S, the log, and the authenticators to a third party, who can repeat Alice's checks and independently verify that a fault has occurred.



(a) Symmetric multi-party scenario (online game)



(b) Asymmetric multi-party scenario (web service)

Figure 2: Multi-party scenarios. The scenario on the left represents a multi-player game; each player is running the game client on his local machine and wants to know whether any other players are cheating. The scenario on the right represents a hosted web service: Alice's software is running on Bob's machine, but the software typically interacts with users other than Alice, such as Alice's customers.

This generic methodology meets our previously stated goals: Alice can detect faults on M, she can obtain evidence, and a third party can check the evidence without having to trust either Alice or Bob.

3.4 Does the AVMM have to be trusted?

A perhaps surprising consequence of this approach is that *the AVMM does not have to be trusted by Alice*. Suppose Bob is malicious and secretly tampers with Alice's software and/or the AVMM, causing M to become faulty. Bob cannot prevent Alice from detecting this: if he tampers with M's log, Alice can tell because the log will not match the authenticators; if he does not, Alice obtains the exact sequence of observable messages M has sent and received, and since by our definition of a fault there is *no* correct execution of M_R that is consistent with this sequence, deterministic replay inevitably fails, no matter what the AVMM recorded.

3.5 Must Alice check the entire log?

For many applications, including the game we consider in this paper, it is perfectly feasible for Alice to audit M's entire log. However, for long-running, compute-intensive applications, Alice may want to save time by doing spot checks on a few log segments instead. The AVMM can enable her to do this by periodically taking a snapshot of the AVM's state. Thus, Alice can independently inspect any segment that begins and ends at a snapshot.

Spot checking sacrifices the completeness of fault detection for efficiency. If Alice chooses to do spot checks, she can only detect faults that manifest as incorrect state transitions in the segments she inspects. An incorrect state transition in an unchecked segment, on the other hand, could permanently modify M's state in a way that is not detectable by checking subsequent segments. Therefore, Alice must be careful when choosing an appropriate policy.

Alice could inspect a random sample of segments plus any segments in which a fault could most likely have a long-term effect on the AVM's state (e.g., during initialization, authentication, key generation). Or, she could inspect segments when she observes suspicious results, starting with the most recent segment and working backwards in reverse chronological order. Spot-checking is most effective in applications where the faults of interest likely occur repeatedly and a single instance causes limited harm, where the application state is frequently re-initialized (preventing long-term effects of a single undetected fault on the state), or where the threat of probabilistic detection is strong enough to deter attackers.

3.6 Do AVMs work with multiple parties?

So far, we have focused on a simple two-party scenario; however, AVMs can be used in more complex scenarios. Figure 2 shows two examples. In the scenario on the left, the players in an online multi-player game are using AVMs to detect whether someone is cheating. Unlike the basic scenario in Figure 1, this scenario is symmetric in the sense that each player is both running software *and* is interested in the correctness of the software on all the other machines. Thus, the roles of auditor and auditee can be played by different parties at different times. The scenario on the right represents a hosted web service: the software is controlled and audited by Alice, but the software typically interacts with parties other than Alice, such as Alice's customers.

For clarity, we will explain our system mostly in terms of the simple two-party scenario in Figure 1. In Section 4.6, we will describe differences for the multi-party case.

4 AVMM design

To demonstrate that AVMs are practical, we now present the design of a specific AVMM.

4.1 Assumptions

Our design relies on the following assumptions:

1. All transmitted messages are eventually received, if retransmitted sufficiently often.
2. All parties (machines and users) have access to a hash function that is pre-image resistant, second pre-image resistant, and collision resistant.
3. Each party has a certified keypair, which can be used to sign messages. Neither signatures nor certificates can be forged.
4. If a user needs to audit the log of a machine, the user has access to a reference copy of the VM image that the machine is expected to use.

The first two are common assumptions made about practical distributed systems. In particular, the first assumption is required for liveness, otherwise it could be impossible to ever complete an audit. The third assumption could be satisfied by providing each machine with a keypair that is signed by the administrator; it is needed to prevent faulty machines from creating fake identities. The fourth assumption is required so that the auditor knows which behaviors are correct.

4.2 Roadmap

Our design instantiates each of the building blocks we have described in Section 3.2: a VMM, a tamper-evident log, and an auditing mechanism. Here, we give a brief overview; the rest of this section describes each building block in more detail.

For the *tamper-evident log* (Section 4.3), we adapt a technique from PeerReview [21], which already comes with a proof of correctness [22]. We extend this log to also include the VMM's execution trace.

The *VMM* we use in this design (Section 4.4) virtualizes a standard commodity PC. This platform is attractive because of the vast amount of existing software that can run on it; however, for historical reasons, it is harder to virtualize than a more modern platform such as Java or .NET. In addition, interactions between the software and the virtual 'hardware' are much more frequent than, e.g., in Java, resulting in a potentially higher overhead.

For *auditing* (Section 4.5), we provide a tool that authenticates the log, then checks it for tampering, and finally uses deterministic replay to determine whether the contents of the log correspond to a correct execution of M_R. If the tool finds any discrepancy between the events in the log and the events that occur during replay, this indicates a fault. Note that, while events such as thread scheduling may appear nondeterministic to an application, they are in fact deterministic from the VMM's perspective. Therefore, as long as all external events (e.g. timer interrupts) are recorded in the log, even race conditions are reproduced exactly during replay and cannot result in false positives.[1]

4.3 Tamper-evident log

The tamper-evident log is structured as a hash chain; each log entry is of the form $e_i := (s_i, t_i, c_i, h_i)$, where s_i is a monotonically increasing sequence number, t_i a type, and c_i data of the specified type. h_i is a hash value that must be linked to all the previous entries in the log, and yet efficient to create. Hence, we compute it as $h_i = H(h_{i-1} \,||\, s_i \,||\, t_i \,||\, H(c_i))$ where $h_0 := 0$, H is a hash function, and $||$ stands for concatenation.

To detect when Bob's machine M forges incoming messages, Alice signs each of her messages with her own private key. The AVMM logs the signatures together with the messages, so that they can be verified during an audit, but it removes them before passing the messages on to the AVM. Thus, this process is transparent to the software running inside the AVM.

To ensure nonrepudiation, the AVMM attaches an authenticator to each outgoing message m. The authenticator for an entry e_i is $a_i := (s_i, h_i, \sigma(s_i \,||\, h_i))$, where the $\sigma(\cdot)$ operator denotes a cryptographic signature with the machine's private key. M also includes h_{i-1}, so that Alice can recalculate $h_i = H(h_{i-1} \,||\, s_i \,||\, \textsc{send} \,||\, H(m))$ and thus verify that the entry e_i is in fact $\textsc{send}(m)$.

To detect when M drops incoming or outgoing messages, both Alice and the AVMM send an *acknowledgment* for each message m they receive. Analogous to the above, M's authenticator in the acknowledgment contains enough information for the recipient to verify that the corresponding entry is $\textsc{recv}(m)$. Alice's own acknowledgment contains just a signed hash of the corresponding message, which the AVMM logs for Alice. When an acknowledgment is not received, the original message is retransmitted a few times. If Alice stops receiving messages from M altogether, she can only suspect that M has failed.

When Alice wants to audit M, she retrieves a pair of authenticators (e.g., the ones with the lowest and highest sequence numbers) and challenges M to produce the log segment that connects them. She then verifies that the hash chain is intact. Because the hash function is second pre-image resistant, it is computationally infeasible to modify the log without breaking the hash chain. Thus, if M has reordered or tampered with a log entry in that

[1]Ensuring deterministic replay on multiprocessor machines is more difficult. We will discuss this in Section 7.4.

segment, or if it has forked its log, M's hash chain will no longer match its previously issued authenticators, and Alice can detect this using this check.

4.4 Virtual machine monitor

In addition to recording all incoming and outgoing messages to the tamper-evident log, the AVMM logs enough information about the execution of the software to enable deterministic replay.

Recording nondeterministic inputs: The AVMM must record all of the AVM's nondeterministic inputs [8]. If an input is asynchronous, the precise timing within the execution must be recorded, so that the input can be reinjected at the exact same point during replay. Hardware interrupts, for example, fall into this category. Note that wall-clock time is not sufficiently precise to describe the timing of such inputs, since the instruction timing can vary on most modern CPUs. Instead, the AVMM uses a combination of instruction pointer, branch counter, and, where necessary, additional registers [15].

Not all inputs are nondeterministic. For example, the values returned by accesses to the AVM's virtual harddisk need not be recorded. Alice knows the system image that the machine is expected to use, and can thus reconstruct the correct inputs during replay. Also many inputs such as software interrupts are synchronous, that is, they are explicitly requested by the AVM. Here, the timing need not be recorded because the requests will be issued again during replay.

Detecting inconsistencies: The tamper-evident log now contains two parallel streams of information: Message exchanges and nondeterministic inputs. Incoming messages appear in both streams: first as messages, and then, as the AVM reads the bytes in the message, as a sequence of inputs. If Bob is malicious, he might try to exploit this by forging messages or by dropping or modifying a message that was received on M before it is injected into the AVM. To detect this, the AVMM cross-references messages and inputs in such a way that any discrepancies can easily be detected during replay.

Snapshots: To enable spot checking and incremental audits (Section 3.5), the AVMM periodically takes a snapshot of the AVM's current state. To save space, snapshots are incremental, that is, they only contain the state that has changed since the last snapshot. The AVMM also maintains a hash tree over the state; after each snapshot, it updates the tree and then records the top-level value in the log. When Alice audits a log segment, she can either download an entire snapshot or incrementally request the parts of the state that are accessed during replay. In either case, she can use the hash tree to authenticate the state she has downloaded.

Taking frequent snapshots enables Alice to perform fine-grain audits, but it also increases the overhead. However, snapshotting techniques have become very efficient; recent work on VM replication has shown that incremental snapshots can be taken up to 40 times per second [11] and with only brief interruptions of the VM, on the order of a few milliseconds. Accountability requires only infrequent snapshots (once every few minutes or hours), so the overhead should be low.

4.5 Auditing and replay

When Alice wants to audit a machine M, she performs the following three steps. First, Alice obtains a segment of M's log and the authenticators that M produced during the execution, so that the log's integrity can be verified. Second, she downloads a snapshot of the AVM at the beginning of the segment. Finally, she replays the entire segment, starting from the snapshot, to check whether the events in the log correspond to a correct execution of the reference software.

Verifying the log: When Alice wants to audit a log segment $e_i \dots e_j$, she retrieves the authenticators she has received from M with sequence numbers in $[s_i, s_j]$. Next, Alice downloads the corresponding log segment L_{ij} from M, starting with the most recent snapshot before e_i and ending at e_j; then she verifies the segment against the authenticators to check for tampering. If this step succeeds, Alice is convinced that the log segment is genuine; thus, she is left with having to establish that the execution described by the segment is correct.

If M is faulty, Alice may not be able to download L_{ij} at all, or M could return a corrupted log segment that causes verification to fail. In either case, Alice can use the most recent authenticator a_j as evidence to convince a third party of the fault. Since the authenticator is signed, the third party can use a_j to verify that log entries with sequence numbers up to s_j must exist; then it can repeat Alice's audit. If no reply is obtained, Alice will suspect Bob.

Verifying the snapshot: Next, Alice must obtain a snapshot of the AVM's state at the beginning of the log segment L_{ij}. If Alice is auditing the entire execution, she can simply use the original software image S. Otherwise she downloads a snapshot from M and recomputes the hash tree to authenticate it against the hash value in L_{ij}.

Verifying the execution: For the final step, Alice needs three inputs: The log segment L_{ij}, the VM snapshot, and the public keys of M and any users who communicated with M. The audit tool performs two checks on L_{ij}, a syntactic check and a semantic check. The syntactic check determines whether the log itself is well-formed, whereas the semantic check determines whether the information in the log corresponds to a correct execution of M_R.

For the syntactic check, the audit tool checks whether all log entries have the proper format, verifies the cryptographic signatures in each message and acknowledgment, checks whether each message was acknowledged, and checks whether the sequence of sent and received messages corresponds to the sequence of messages that enter and exit the AVM. If any of these tests fail, the tool reports a fault.

For the semantic check, the tool locally instantiates a virtual machine that implements M_R, and it initializes the machine with the snapshot, if any, or S. Next, it reads L_{ij} from beginning to end, replaying the inputs, checking the outputs against the outputs in L_{ij}, and verifying any snapshot hashes in L_{ij} against snapshots of the replayed execution (to be sure that the snapshot at the end of L_{ij} is also correct). If there is any discrepancy whatsoever (for example, if the virtual machine produces outputs that are not in the log, or if it requests the synchronous inputs in a different order), replay terminates and reports a fault. In this case, Alice can use L_{ij} and the authenticators as evidence to convince Bob, or any other interested party, that M is faulty.

If the log segment L_{ij} passes all of the above checks, the tool reports success and then terminates. Auditing can be performed offline (after the execution of a given log segment is finished) or online (while the execution is in progress).

4.6 Multi-party scenario

So far, we have described the AVMM in terms of the simple two-party scenario. A multi-party scenario requires three changes. First, when some user wants to audit a machine M, he needs to collect authenticators from other users that may have communicated with M. In the gaming scenario in Figure 2(a), Alice could download authenticators from Charlie before auditing Bob. In the web-service scenario in Figure 2(b), the users could forward any authenticators they receive to Alice.

Second, with more than two parties, network problems could make the same node appear unresponsive to some nodes and alive to others. Bob could exploit this, for instance, to avoid responding to Alice's request for an incriminating log segment, while continuing to work with other nodes. To prevent this type of attack, Alice forwards the message that M does not answer as a *challenge* for M to the other nodes. All nodes stop communicating with M until it responds to the challenge. If M is correct but there is a network problem between M and Alice, or M was temporarily unresponsive, it can answer the challenge and its response is forwarded to Alice.

Third, when one user obtains evidence of a fault, he may need to distribute that evidence to other interested parties. For example, in the gaming scenario, if Alice detects that Bob is cheating, she can send the evidence to Charlie, who can verify it independently; then both can decide never to play with Bob again.

4.7 Guarantees

Given our assumptions from Section 4.1 and the fault definition from Section 3.1, the AVMM offers the following two guarantees:

- **Completeness:** If the machine M is faulty, a full audit of M will report a fault and produce evidence against M that can be verified by a third party.
- **Accuracy:** If the machine M is *not* faulty, no audit of M will report a fault, and there cannot exist any valid evidence against M.

If Alice performs spot checks on a number of log segments $s_1, \ldots, s_k$ rather than a full audit, accuracy still holds. However, if M is faulty, her audit will only report the fault and produce evidence if there exists at least one log segment s_i in which the fault manifests. These guarantees are independent of the software S, and they hold for any fault that manifests as a deviation from M_R, even if Alice, Bob, and/or other users are malicious. A proof of these properties is presented in Appendix A.

Since our design is based on the tamper-evident log from PeerReview [21], the resulting AVMM inherits a powerful property from PeerReview: in a distributed system with multiple nodes, it is possible to audit the execution of the entire system by auditing each node individually. For more details, please refer to [21].

4.8 Limitations

We note two limitations implied by the AVMM's guarantees. First, AVMs cannot detect bugs or vulnerabilities in the software S, because the expected behavior of M is defined by M_R and thus S. If S has a bug and the bug is exercised during an execution, an audit will succeed. For instance, if S allows unauthorized software modifications, Bob could use this feature to change or replace S. Alice must therefore make sure that S does not have vulnerabilities that Bob could exploit.

Second, any behavior that can be achieved by providing appropriate inputs to M_R is considered correct. When such inputs come from sources other than the network, they cannot be verified during an audit. In some applications, Bob may be able to exploit this fact by recording local (non-network) inputs in the log that elicit some behavior in M_R he desires.

5 Application: Cheat detection in games

AVMs and AVMMs are application-independent, but for our evaluation, we focus on one specific application, namely cheat detection. We begin by characterizing the class of cheats that AVMs can detect, and we discuss how AVMs compare to the anti-cheat systems that are in use today.

5.1 How are cheats detected today?

Today, many online games use anti-cheating systems like PunkBuster [35], the Warden [23] or Valve Anti-Cheat (VAC) [38]. These systems work by scanning the user's machine for known cheats [23, 24, 35]; some allow the game admins to request screenshots or to perform memory scans. In addition to privacy concerns, this approach has led to an arms race between cheaters and game maintainers, in which the former constantly release new cheats or variations of existing ones, and the latter must struggle to keep their databases up to date.

5.2 How can AVMs be used with games?

Recall that AVMs run entire VM images rather than individual programs. Hence, the players first need to agree on a VM image that they will use. For example, one of them could install an operating system and the game itself in a VM, create a snapshot of the VM, and then distribute the snapshot to the other players. Each player then initializes his AVM with the agreed-upon snapshot and plays while recording a log. If a player wishes to reassure himself that other players have not cheated, he can request their logs (during or after the game), check them for tampering, and replay them using his own, trusted copy of the agreed-upon VM image.

Since many cheats involve installing additional programs or modifying existing ones, it is important to disable software installation in the snapshot that is used during the game, e.g., by revoking the necessary privileges from all accounts that are accessible to the players. Otherwise, downloading and installing a cheat would simply be re-executed during replay without causing any discrepancies. However, note that this restriction is only required *during* the game; it does not prevent the maintainer of the original VM image from installing upgrades or patches.

5.3 How do players cheat in games?

Players can cheat in many different ways – a recent taxonomy [41] identified no less than fifteen different types of cheats, including collusion, denial of service, timing cheats, and social engineering. In Section 5.4, we discuss which of these cheats AVMs are effective against, and we illustrate our discussion with three concrete examples of cheats that are used in Counterstrike. Since the reader may not be familiar with these cheats, we describe them here first.

The first cheat is an *aimbot*. Its purpose to help the cheater with target acquisition. When the aimbot is active, the cheater only needs to point his weapon in the approximate direction of an opponent; the aimbot then automatically aims the weapon exactly at that opponent. An aimbot is an example of a cheat that works, at least conceptually, by feeding the game with forged inputs.

The second cheat is a *wallhack*. Its purpose is to allow the cheater to see through opaque objects, such as walls. Wallhacks work because the game usually renders a much larger part of the scenery than is actually visible on screen. Thus, if the textures on opaque objects are made transparent or removed entirely, e.g., by a special graphics driver [37], the objects behind them become visible. A wallhack is an example of a cheat that violates secrecy; it reveals information that is available to the game but is not meant to be displayed.

The third cheat is *unlimited ammunition*. The variant we used identifies the memory location in the Counterstrike process that holds the cheater's current amount of ammunition, and then periodically writes a constant value to that location. Thus, even if the cheater constantly fires his weapon, he never runs out (similar cheats exist for other resources, e.g., unlimited health). This cheat changes the network-visible behavior of the cheater's machine. It is representative of a larger class of cheats that rely on modifying local in-memory state; other examples include teleportation, which changes the variable that holds the player's current position, or unlimited health.

5.4 Which cheats can AVMs detect?

AVMs are effective against two specific, broad classes of cheats, namely

1. cheats that need to be installed along with the game in some way, e.g., as loadable modules, patches, or companion programs; and
2. cheats that make the network-visible behavior of the cheater's machine inconsistent with any correct execution.

Both types of cheats cause replay to fail when the cheater's machine is audited. In the first case, the reason is that replay can only succeed if the VM images used during recording and replay produce the same sequence of events recorded in the log. If different code is executed or different data is read at any time, replay almost certainly diverges soon afterward. In the second case, replay fails by definition because there exists *no* correct execution that is consistent with the network traffic the cheater's machine has produced.

If a cheat is in the first class but not in the second, it may be possible to re-engineer it to avoid detection. Common examples include cheats that violate secrecy,

Total number of cheats examined	26
Cheats detectable with AVMs	26
... in this specific implementation of the cheat	22
... no matter how the cheat is implemented	4
Cheats not detectable with AVMs	0

Table 1: Detectability of Counterstrike cheats from popular Counterstrike discussion forums

such as wallhacks, and cheats that rely on forged inputs, such as aimbots. For instance, a cheater might implement an aimbot as a separate program that runs outside of the AVM and aims the player's weapon by feeding fake inputs to the AVM's USB port. A particularly tech-savvy cheater might even set up a second machine that uses a camera to capture the game state from the first machine's screen and a robot arm to type commands on the first machine's keyboard. While such cheats are by no means impossible, they do require substantially more effort and expertise than a simple patch or module that manipulates the game state directly. Thus, AVMs raise the bar significantly for such cheats.

In contrast, cheats in the second class can be detected by AVMs in *any* implementation. Examples of such cheats include unlimited ammunition, unlimited health, or teleportation. For instance, if a player has k rounds of ammunition and uses a cheat of any type to fire more than k shots, replay inevitably fails because there is *no* correct execution of the game software in which a player can fire after having run out of ammunition. AVMs are effective against any current or future cheats that fall into this category.

We hypothesize that the first class includes almost all cheats that are in use today. To test this hypothesis, we downloaded and examined 26 real Counterstrike cheats from popular discussion forums on the Internet (Table 1). We found that every single one of them had to be installed in the game AVM to be effective, and would therefore be detected. We also found that at least 4 of the 26 cheats additionally belonged to the second class and could therefore be detected not only in their current form, but also in any future implementation.

5.5 Summary

Even though we did not specifically design AVMs for cheat detection, they do offer three important advantages over current anti-cheating solutions like VAC or PunkBuster. First, they protect players' privacy by separating auditable computation (the game in the AVM) from non-auditable computation (e.g., browser or banking software running outside the AVM). Second, they are effective against virtually all current cheats, including novel, rare, or unknown cheats. Third, they are guaranteed to detect all possible cheats of a certain type, no matter how they are implemented.

6 Evaluation

In this section, we describe our AVMM prototype, and we report how we used it to detect cheating in Counterstrike, a popular multi-player game. Our goal is to answer the following three questions:

1. Does the AVMM work with state-of-the-art games?
2. Are AVMs effective against real cheats?
3. Is the overhead low enough to be practical?

6.1 Prototype implementation

Our prototype AVMM implementation is based on VMware Workstation 6.5.1, a state-of-the-art virtual machine monitor whose source code we obtained through VMware's Academic Program. VMware Workstation supports a wide range of guest operating systems, including Linux and Microsoft Windows, and its VMM already supports many features that are useful for AVMs, such as deterministic replay and incremental snapshots. We extended the VMM to record extra information about incoming and outgoing network packets, and we added support for tamper-evident logging, for which we adapted code from PeerReview [21]. Since VMware Workstation only supports uniprocessor replay, our prototype is limited to AVMs with a single virtual core (see Section 7.4 for a discussion of multiprocessor replay). However, most of the logging functionality is implemented in a separate daemon process that communicates with the VMM through kernel-level pipes, so the AVMM can take advantage of multi-core CPUs by using one of the cores for logging, cryptographic operations and auditing, while running AVMs on the other cores at full speed.

Our audit tool implements a two-step process: Players first perform the syntactic check using a separate tool and then run the semantic check by replaying the log in a local AVM, using a copy of the VM image they trust. If at least one of the two stages fails, they can give the log and the authenticators as evidence to fellow players—or, indeed, any third party. All steps are deterministic, so the other party will obtain the same result.

6.2 Experimental setup

For our evaluation, we used the AVMM prototype to detect cheating in Counterstrike. There are two reasons for this choice. First, Counterstrike is played in a variety of online leagues, as well as in worldwide championships such as the World Cyber Games, which makes cheating a matter of serious concern. Second, there is a large

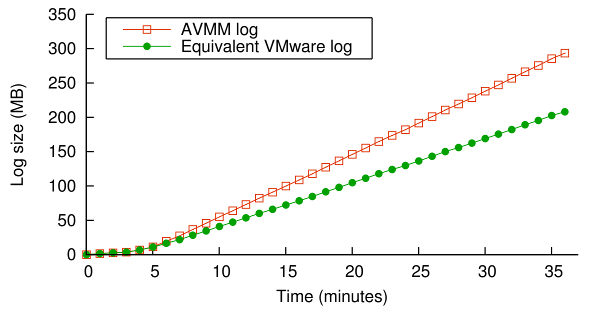


Figure 3: Growth of the AVMM log, and an equivalent VMware log, while playing Counterstrike.

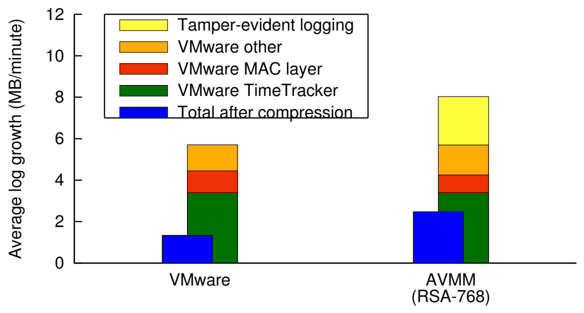


Figure 4: Average log growth for Counterstrike by content. The bars in front show the size after compression.

and diverse ecosystem of readily available Counterstrike cheats, which we can use for our experiments.

Our experiments are designed to model a Counterstrike game as it would be played at a competition or at a LAN party. We used three Dell Precision T1500 workstations, one for each player, with 8 GB of memory and 2.8 GHz Intel Core i7 860 CPUs. Each CPU has four cores and two hyperthreads per core. The machines were connected to the same switch via 1 Gbps Ethernet links, and they were running Linux 2.6.32 (Debian 5.0.4) as the host operating system. On each machine, we installed an AVMM binary that was based on a VMware Workstation 6.5.1 release build. Each player had access to an 'official' VM snapshot, which contained Windows XP SP3 as the guest operating system, as well as Counterstrike 1.6 at patch version 1.1.2.5. Sound and voice were disabled in the game and in VMware. As discussed in Section 5.2, we configured the snapshot to disallow software installation. In the snapshot, the OS was already booted, and the player was logged in without administrator privileges.

All players were using 768-bit RSA keys. These keys are not strong enough to provide long-term security, but in our scenario the signatures only need to last until any cheaters have been identified, i.e., at most a few days or weeks beyond the end of the game. In December 2009, factoring a 768-bit number took almost 2,000 Opteron-CPU years [3], so this key length should be safe for gaming purposes for some time to come.

To quantify the costs of various aspects of AVMs, we ran experiments in five different configurations. **bare-hw** is our baseline configuration in which the game runs directly on the hardware, without virtualization. **vmware-norec** adds the virtual machine monitor without our modifications, and **vmware-rec** adds the logging for deterministic replay. **avmm-nosig** uses our AVMM implementation without signatures, and **avmm-rsa768** is the full system as described.

We removed the default frame rate cap of 72 fps, so that Counterstrike rendered frames as quickly as the available CPU resources allow and we can use the achieved frame rate as a performance metric. In Section 6.5 we consider a configuration with the default frame rate cap. To make sure the performance of bare-hw and virtualized configurations can be compared, we configured the game to run without OpenGL, which is not supported in our version of VMware Workstation, and we ran the game in window rather than full-screen mode. We played each game for at least thirty minutes.

6.3 Functionality check

Recall from Section 5.4 that AVMs can detect by design all of the 26 cheats we examined. As a sanity check to validate our implementation, we tried four Counterstrike cheats in our collection that do not depend on OpenGL. For each cheat, we created a modified VM image that had the cheat preinstalled, and we ran an experiment in the avmm-rsa768 configuration where one of the players used the special VM image and activated the cheat. We then audited each player; as expected, the audits of the honest players all succeeded, while the audits of the cheater failed due to a divergence during replay.

6.4 Log size and contents

The AVMM records a log of the AVM's execution during game play. To determine how fast this log grows, we played the game in the avmm-rsa768 configuration, and we measured the log size over time. Figure 3 shows the results. The log grows slowly while players are joining the game (until about 3 minutes into the experiment) and then continues to grow steadily during game play, by about 8 MB per minute. For comparison, we also show the size of an equivalent VMware log; the difference is due to the extra information that is required to make the log tamper-evident.

Figure 4 shows the average log growth rate about the content. More than 70% of the AVMM log consist of information needed for replay; tamper-evident logging is responsible for the rest. The replay information consists mainly of `TimeTracker` entries (59%), which are used by the VMM to record the exact timing of events, and MAC-layer entries (14%), such as incoming or outgoing network packets; other entry types account for the remaining 27%. The composition of the VMware log differs slightly because the packet payloads are stored in the MAC-layer entries rather than in the tamper-evident logging entries. We also show results after applying `bzip2` and a lossless, VMM-specific (but application-independent) compression algorithm we developed. This brings the average log growth rate to 2.47 MB per minute.

From these results, we can estimate that a one-hour game session would result in a 480 MB log, or 148 MB after compression. Thus, given that current hard disk capacities are measured in terabytes, storage should not be a problem, even for very long games. Also, when a player is audited, he must upload his log to his fellow players. If the game is played over the Internet, uploading a one-hour log would take about 21 minutes over a 1 Mbps upstream link. If the game is played over a LAN, e.g., at a competition, the upload would complete in a few seconds. To avoid detection delays, our prototype can also perform auditing concurrently with the game; we evaluate this feature in Section 6.11.

6.5 Low growth with the frame rate cap

Recall that Counterstrike was configured without a frame rate cap in our experiments, so that the measured frame rate can be used as a performance metric. We discovered that when the frame rate cap is enabled, Counterstrike appears to implement inter-frame delays by busy-waiting in a tight loop, reading the system clock. Since the AVMM has to log every clock access as a nondeterministic input, this increases the log growth considerably—by a factor of 18 when the default cap of 72 fps is used.

To reduce the log growth for applications that exhibit this behavior, we experimented with the following optimization. Whenever the AVMM observes consecutive clock reads from the same AVM within $5\,\mu$s of each other, it delays the n.th consecutive read by $2^{n-2}*50\,\mu$s, starting with the second read and up to a limit of 5 ms. The exponential progression of delays limits the number of clock reads during long waits, but does not unduly affect timing accuracy during short waits.

This optimization is very effective: log growth is actually 2% lower than reported in Section 6.4, with or without the frame-rate cap. Moreover, the uncapped frame rate is only 3% lower than the rate without the optimization, which shows that the optimization has only a mild impact on game performance.

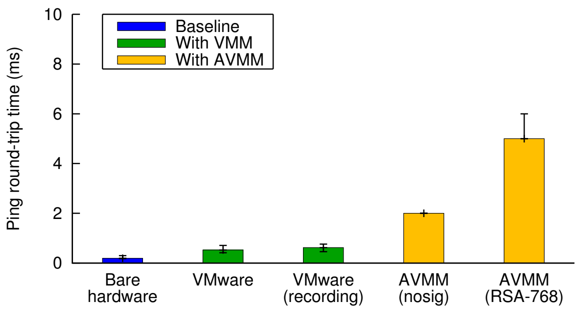


Figure 5: Median ping round-trip times. The error bars show the 5th and the 95th percentile.

6.6 Syntactic and semantic check

Alice can audit another player Bob by checking Bob's log against his authenticators (syntactic check) and by replaying Bob's log using a trusted copy of the VM image (semantic check). We expect the syntactic check to be relatively fast, since it is essentially a matter of verifying signatures, whereas the replay involves repeating all the computations that were performed during the original game and should therefore take about as long as the game itself. Our experiments with the log of the server machine from the avmm-rsa768 configuration (which covers 2,216 seconds with 1,987 seconds of actual game play) confirm this. We needed 34.7 seconds to compress the log, 13.2 seconds to decompress it, 6.9 seconds for the syntactic check, and 1,977 seconds for the semantic check (2,031 seconds total). Replay was actually a bit faster because the AVMM skips any time periods in the recording during which the CPU was idle, e.g., before the game was started.

Unlike the performance of the actual game, the speed of auditing is not critical because it can be performed at leisure, e.g., in the background while the machine is used for something else.

6.7 Network traffic

The AVMM increases the amount of network traffic for two reasons: First, it adds a cryptographic signature to each packet, and second, it encapsulates all packets in a TCP connection. To quantify this overhead, we measured the raw, IP-level network traffic in the bare-hw configuration and in the avmm-rsa768 configuration. On average, the machine hosting the game sent 22 kbps in bare-hw and 215.5 kbps in avmm-rsa768.

This high relative increase is partly due to the fact that Counterstrike clients send extremely small packets of 50–60 bytes each, at 26 packets/sec, so the AVMM's fixed per-packet overhead (which includes one cryptographic signature for each packet and one for each acknowledgment) has a much higher impact than it would

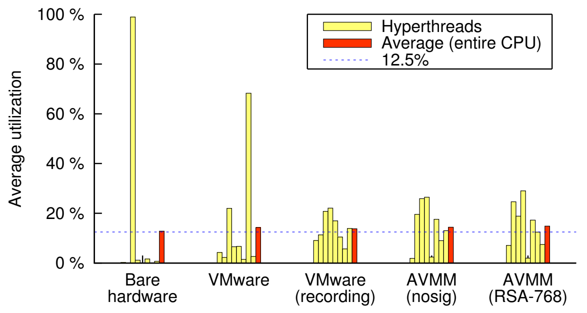


Figure 6: Average CPU utilization in Counterstrike for each of the eight hyperthreads, and for the entire CPU.

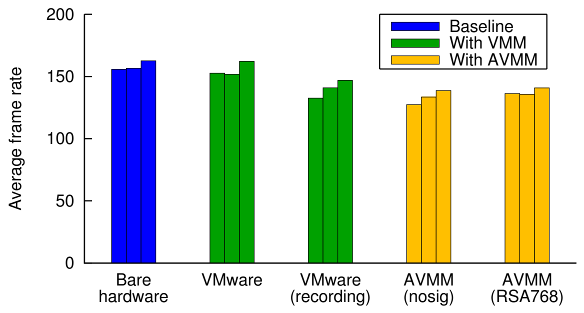


Figure 7: Frame rate in Counterstrike for each of the three machines. The left machine was hosting the game.

for packets of average Internet packet size. However, in absolute terms, the traffic is still quite low and well within the capabilities of even a slow broadband upstream.

6.8 Latency

The AVMM adds some latency to packet transmissions because of the logging and processing of authenticators. To quantify this, we ran an AVM in five different configurations and measured the round-trip time (RTT) of 100 ICMP Echo Request packets. Figure 5 shows the median RTT, as well as the 5th and the 95th percentile. Since our machines are connected to the same switch, the RTT on bare hardware is only 192 μs; adding virtualization increases it to 525 μs, VMware recording to 621 μs, and the daemon to above 2 ms. Enabling 768-bit RSA signatures brings the total RTT to about 5 ms. Recall that both the ping and the pong are acknowledged, so four signatures need to be generated and verified. Since the critical threshold for interactive applications is well above 100 ms [12], 5 ms seem tolerable for games. The overhead could be reduced by using a signing algorithm such as ESIGN [34], which can generate and verify a 2046-bit signature in less than 125 μs.

6.9 CPU utilization

Compared to a Counterstrike game on bare hardware, the AVMM requires additional CPU power for virtualization and for the tamper-evident log. To quantify this overhead, we measured the CPU utilization in five configurations, ranging from bare-hw to avmm-rsa768. To isolate the contribution from the tamper-evident log, we pinned the daemon process to hyperthread 0 (HT 0) in the AVMM experiments and restricted the game to the other hyperthreads except for HT 0's hypertwin, HT 4, which shares a core with HT 0.[2] One of the machines in our experiments runs the Counterstrike server in addition to serving a player. To be conservative, we report numbers for this machine, as it has the highest load.

Figure 6 shows the average utilization for each HT, as well as the average across the entire CPU. The utilization of HT 0 (below 8%) in the AVM experiments shows that the overhead from the tamper-evident log is low. The game is constantly busy rendering frames, but because the Counterstrike rendering engine is single-threaded, it cannot run on more than one HT at a time. The OS/VMM will sometimes schedule it on one HT and sometimes on another, thus we expect an average utilization over the eight HTs of 12.5%, which our results confirm.

6.10 Frame rate

Since the game is rendering frames as fast as the available CPU cycles allow, a meaningful metric for the CPU overhead is the achieved frame rate, which we consider next. To measure the frame rate, we wrote an AMX Mod X [1] script that increments a counter every time a frame is rendered. We read out this counter at the beginning and at the end of each game, and we divided the difference by the elapsed time. Figure 7 shows our results for each of the three machines. The results vary over time and among players, because the frame rate depends on the complexity of the scene being rendered, and thus on the path taken by each player.

The frame rate on the AVMM is about 13% lower than on bare hardware. The biggest overhead seems to come from enabling recording in VMware Workstation, which causes the average frame rate to drop by about 11%. In absolute terms, the resulting frame rate (137 fps) is still very high; posts in Counterstrike forums generally recommend configuring the game for about 60–80 fps.

To quantify the advantage of running some of the AVMM logic on a different HT, we ran an additional experiment with both Counterstrike and all AVMM threads pinned to the same hyperthread. This reduced the average frame rate by another 11 fps.

[2]Nevertheless, the load on HT 4 is not exactly zero because Linux performs kernel-level IRQ handling on lightly-loaded hyperthreads.

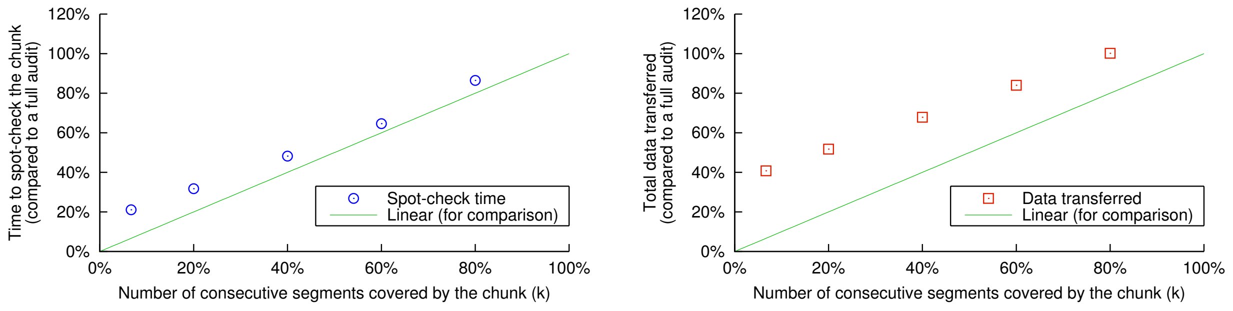


Figure 9: Efficiency of spot checking. The cost of a spot check is roughly proportional to the size of the checked chunk, but there is a fixed cost per chunk for transferring the memory and disk snapshots and for data decompression.

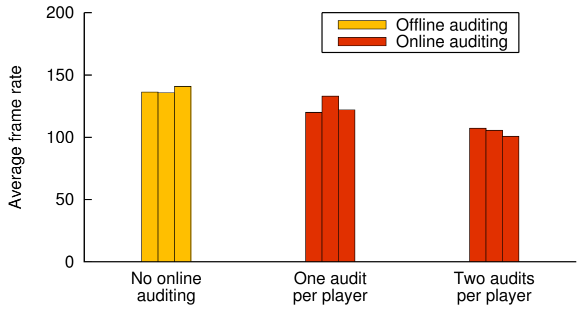


Figure 8: Frame rate for each of the three machines with zero, one, or two online audits per machine.

6.11 Online auditing

If a game session is long or the stakes are particularly high, players may wish to detect cheaters well before the end of the game. In such cases, players can incrementally audit other players' logs while the game is still in progress. In this configuration, which we refer to as *online auditing*, cheating could be detected as soon as the externally visible behavior of the cheater's machine deviates from that of the reference machine.

If a player uses the same machine to concurrently play the game and audit other players, the higher resource consumption can affect game performance. To quantify this effect, we played the game in the avmm-rsa768 configuration with each player auditing zero, one, or two other players on the same machine. As before, we measured the average frame rate experienced by each player.

Figure 8 shows our results. With an increasing number of players audited, the frame rate drops somewhat, from 137 fps with no audits to 104 fps with two audits. However, the drop is less pronounced than expected because the audits can leverage the unused cores. If the number of audits a is increased further, we expect the game performance to eventually degrade with $1/a$.

Since replay is slightly slower than the original execution, auditing falls behind the game by about four seconds per minute of play, even when the audit executes on an otherwise unloaded machine. To ensure quick detection even during very long game sessions, we can compensate by artificially slowing down the original execution. We found that a 5% slowdown was sufficient to allow the auditor to keep up; this reduced the frame rate by up to 7 fps. Note that a certain lag can actually be useful to prevent players from learning each other's current positions or strategies through an audit. In practice, players may want to disallow audits of the current round and/or the most recent moments of game play.

6.12 Spot checking

Online games are not a very interesting use case for spot checking because complete audits are feasible. Therefore, we set up a simple additional experiment that models a client/server system – specifically, a MySQL 5.0.51 server in one AVM and a client running MySQL's `sql-bench` benchmark in another. We ran this experiment for 75 minutes in the avmm-rsa768 configuration, and we recorded a snapshot every five minutes. We found that, on average, our prototype takes 5 seconds to record a snapshot. The incremental disk snapshots are between 1.9 MB and 91 MB, while each memory snapshot occupies about 530 MB. The reason for the latter is that VMware Workstation creates a full dump of the AVM's main memory (512 MB) for each snapshot. This could probably be optimized considerably, e.g., using techniques from Remus [11].

In the following, we refer to the part of the log between two consecutive snapshots as a *segment*, and to k consecutive segments as a k*-chunk*. To quantify the costs of spot checking, we audited all possible k-chunks in our log for $k \in \{1, 3, 5, 9, 12\}$, and measured the amount of data that must be transferred over the network, as well as the time it takes to replay the chunk. However, we excluded k-chunks that start at the beginning of the log; these are atypical because a) they are the only chunks for which no memory or disk snapshots have to be transferred, and b) they have less activity because the MySQL server is not yet running at the beginning. We report averages because the results for chunks with the same value of k never varied by more than 10%.

Figure 9 shows the results, normalized to the cost of a full audit. As expected, the cost grows with the chunk size k; however, there is an additional fixed cost per chunk for transferring the corresponding memory and disk snapshots.

6.13 Summary

Having reported our results, we now revisit our three initial questions. We have demonstrated that our AVMM works out-of-the-box with Counterstrike, a state-of-the-art game, and we have shown that it is effective against real cheats we downloaded from Counterstrike forums on the Internet. AVMs are not free; they affect various metrics such as latency, traffic, or CPU utilization, and they reduce the frame rate by about 13%, compared to the rate achieved on bare hardware. In return for this overhead, players gain the ability to audit other players. Auditing takes time, in some cases as much as the game itself, but it seems time well spent because it either exposes a cheater or clears an innocent player of suspicion. AVMs provide this novel capability by combining two seemingly unrelated technologies, tamper-evident logs and virtualization.

7 Discussion

7.1 Other applications

AVMs are application-independent and can be used in applications other than games.

Distributed systems: AVMs can be used to make any distributed system accountable, simply by executing the software on each node within an AVM. The node software can be arbitrarily complex and available only as a binary system image. Accountability is useful in distributed systems where principals have an interest in monitoring the behavior of other principals' nodes, and where post factum detection is sufficient. Such systems include federated systems where no single entity has complete control or visibility of the entire system, where different parties compete (e.g., in an online game, an auction, or a federated system like the Internet) or where parties are expected to cooperate but lack adequate incentives to do so (e.g., in a peer-to-peer system).

Network traffic accountability: AVMs could also be useful in detecting advanced forms of malware that could escape online detection mechanisms. An AVM, combined with a traffic monitor that records a machine's network communication, can capture the network-observable behavior of a machine, and replay it later with expensive intrusion detection (e.g., taint tracking) in place.

Cloud computing: Another potential application of AVMs is cloud computing. AVMs can enable cloud customers to verify that their software executes in the cloud as expected. AVM are a perfect match for infrastructure-as-a-service (IaaS) clouds that offer customers a virtual machine. However, AVMs in the cloud face additional challenges: auditors cannot easily replay the entire execution for lack of resources; accountable services must be able to interact with non-accountable clients; and, it may not be practical to sign every single packet. The first challenge can be addressed with spot checking (Section 3.5). We plan to address the remaining challenges in future work.

7.2 Using trust to get stronger guarantees

One of the strengths of AVMs is that they can verify the integrity of a remote node's execution without relying on trusted components. However, if trusted components are available, we can obtain additional guarantees. We sketch two possible extensions below.

Secure local input: AVMs cannot detect the hypothetical re-engineered aimbot from Section 5.4 because existing hardware does not authenticate events from local input devices, such as keyboards or mice. Thus, a compromised AVMM can forge or suppress local inputs, and even a correct AVMM cannot know whether a given keystroke was generated by the user or synthesized by another program, or another machine. This limitation can be overcome by adding crypto support to the input devices. For example, keyboards could sign keystroke events before reporting them to the OS, and an auditor could verify that the keystrokes are genuine using the keyboard's public key. Since most peripherals generate input at relatively low rates, the necessary hardware should not be expensive to build.

Trusted AVMM: If we can trust the AVMM that is running on a remote node, we can detect additional classes of cheats and attacks, including certain attacks on confidentiality. For example, a trusted AVMM could establish a secure channel between the AVM and Alice (even if the software in the AVM does not support encryption) and thus prevent Bob's machine from leaking information by secretly communicating with other machines. A trusted AVMM could also prevent wallhacks (see Section 5.3) by controlling outside access to the machine's graphics card. If trusted hardware, such as memory encryption [40] is available on Bob's machine, the AVMM could even prevent Bob from reading information directly from memory. Remote attestation could be used to make sure that a trusted AVMM is indeed running on a remote computer, e.g., using a system like Terra [17].

7.3 Accountability versus privacy

Ideally, an accountability system should disclose to an auditor only the information strictly required to determine that the auditee has met his obligations. By this standard, AVM logs are rather verbose: an AVM records enough information to replay the execution of the software it is running. This is a price we pay for the generality of AVMs—they can detect a large class of faults in complex software available only in binary form. In practice, the amount of extra information released can be controlled.

Let us consider how the extra information captured in the AVM logs affect Alice and Bob's privacy. The log reveals information about actions of Bob's machine, but only about the execution inside a given AVM, and only to approved auditors. In the web service scenario (Figure 2b), Alice is presumably paying Bob for running her software in an AVM, so she has every right to know about the execution of the software. Similarly, it is not unreasonable to expect players in a game to share information about their game execution. In either case, the auditor cannot observe executions the auditee may be running outside the audited AVM.

Alice and Bob's privacy may be affected when she uses part of the log as evidence to demonstrate a fault on Bob's machine to a third party. The evidence reveals additional information about the AVM, including a snapshot, to that party. Therefore, Alice should release evidence only to third parties that have a legitimate need to know about faults on Bob's machine. To limit the extra information released to third parties, Alice can use the hash tree (Section 4.4) to remove any part of the snapshot that is not necessary to replay the relevant segment.

7.4 Replay for multiprocessors

Our prototype AVMM can assign only a single CPU core for each AVM, because VMware's deterministic replay is limited to uniprocessors. SMP-ReVirt [16] has recently demonstrated that deterministic replay is also possible for multiprocessors, but its cost is substantially higher than the cost of uniprocessor replay. Because replay is a building block for many important applications, such as forensics [15], replication [11], and debugging [25], there is considerable interest in developing more efficient techniques [5, 13, 16, 28, 29]. As more efficient techniques become available, AVMMs can directly benefit from them.

7.5 Bug detection

Recall that AVMs define faults as deviations from the behavior of a reference implementation. If the reference implementation has a bug and this bug is triggered during an execution, it will behave identically during the replay, and thus it will not be classified as a fault. If a bug in the reference implementation permits unauthorized software modification (e.g., a buffer overflow bug), then neither the modification itself nor the behavior of the modified software will be reported as a fault.

Detecting bugs in the reference implementation is outside the fault model AVMs were designed to detect. However, deterministic execution replay provides an opportunity to use sophisticated runtime analysis tools during auditing [10]. In particular, techniques whose runtime costs are too high for deployment in a live system could be used during an off-line replay. Taint tracking, for instance, can reliably detect the unsafe use of data that were received from an untrusted source [33], thus detecting buffer overwrite attacks and other forms of unauthorized software installation. More generally, sophisticated runtime techniques can be used during replay to detect bugs, vulnerabilities and attacks as part of a normal audit.

8 Conclusion

Accountable virtual machines (AVM) allow users to audit software executing on remote machines. An AVM can detect a large and general class of faults, and it produces evidence that can be verified independently by a third party. At the same time, an AVM allows the operator of the remote machine to prove whether his machine is correct. To demonstrate that AVMs are feasible, we have designed and implemented an AVM monitor based on VMware Workstation and used it to detect real cheats in Counterstrike, a popular online multiplayer game. Players can record their game execution in a tamper-evident manner at a modest cost in frame rate. Other players can audit the execution to detect cheats, either after the game has finished or concurrently with the game. The system is able to detect all of 26 existing cheats we examined.

Acknowledgments

We appreciate the detailed and helpful feedback from Jon Howell, the anonymous OSDI reviewers, and our shepherd, Mendel Rosenblum. We would like to thank VMware for making the source code of VMware Workstation available to us under the VMware Academic Program, and our technical contact, Jim Chow, who has been extremely helpful. Finally, we would like to thank our many enthusiastic Counterstrike volunteers.

References

[1] AMX Mod X project. `http://www.amxmodx.org/`.

[2] D. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet protocol (AIP). In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, Aug. 2008.

[3] K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmerman. Factorization of a 768-bit RSA modulus. `http://eprint.iacr.org/2010/006.pdf`.

[4] K. Argyraki, P. Maniatis, O. Irzak, and S. Shenker. An accountability interface for the Internet. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, Oct. 2007.

[5] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, Oct. 2010.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[7] N. E. Baughman, M. Liberatore, and B. N. Levine. Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Transactions on Networking (ToN)*, 15(1):1–13, Feb. 2007.

[8] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[9] C. Chambers, W. Feng, W. Feng, and D. Saha. Mitigating information exposure to cheaters in real-time strategy games. In *Proceedings of the ACM International Workshop on Network and operating systems support for digital audio and video (NOSSDAV)*, June 2005.

[10] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the USENIX Annual Technical Conference*, June 2008.

[11] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2008.

[12] J. Dabrowski and E. V. Munson. Is 100 milliseconds too fast? In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, Apr. 2001.

[13] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2009.

[14] R. Dingledine, M. J. Freedman, and D. Molnar. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter Accountability. O'Reilly and Associates, 2001.

[15] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2002.

[16] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman. Execution replay for multiprocessor virtual machines. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, Mar. 2008.

[17] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[18] A. Haeberlen. A case for the accountable cloud. In *Proceedings of the ACM SIGOPS International Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, Oct. 2009.

[19] A. Haeberlen, I. Avramopoulos, J. Rexford, and P. Druschel. NetReview: Detecting when interdomain routing goes wrong. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009.

[20] A. Haeberlen, P. Kuznetsov, and P. Druschel. The case for Byzantine fault detection. In *Proceedings of HotDep*, Nov. 2006.

[21] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[22] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. Technical Report 2007-3, Max Planck Institute for Software Systems, Oct. 2007.

[23] G. Hoglund. 4.5 million copies of EULA-compliant spyware. `http://www.rootkit.com/blog.php?newsid=358`.

[24] G. Hoglund and G. McGraw. *Exploiting Online Games: Cheating Massively Distributed Systems*. Addison-Wesley, 2007.

[25] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, Apr. 2005.

[26] B. W. Lampson. Computer security in the real world. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2000.

[27] P. Laskowski and J. Chuang. Network monitors and contracting systems: competition and innovation. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, Sept. 2006.

[28] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline symbolic analysis for multi-processor execution replay. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2009.

[29] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2010.

[30] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009.

[31] N. Michalakis, R. Soulé, and R. Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2007.

[32] C. Mönch, G. Grimen, and R. Midtstraum. Protecting online games against cheating. In *Proceedings of the Workshop on Network and Systems Support for Games (NetGames)*, Oct. 2006.

[33] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Annual Network and Distributed Systems Security Symposium (NDSS)*, Feb. 2005.

[34] T. Okamoto. A fast signature scheme based on congruential polynomial operations. *IEEE Transactions on Information Theory*, 36(1):47–53, 1990.

[35] PunkBuster web site. `http://www.evenbalance.com/`.

[36] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[37] A. Smith. ASUS releases games cheat drivers. `http://www.theregister.co.uk/2001/05/10/asus_releases_games_cheat_drivers/`, May 2001.

[38] Valve Corporation. Valve anti-cheat system (VAC). `https://support.steampowered.com/kb_article.php?ref=7849-RADZ-6869`.

[39] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Annual Workshop on Modeling, Benchmarking, and Simulation (MoBS)*, June 2007.

[40] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin. Improving cost, performance, and security of memory encryption and authentication. *ACM SIGARCH Computer Architecture News*, 34(2):179–190, 2006.

[41] J. Yan and B. Randell. A systematic classification of cheating in online games. In *Proceedings of the Workshop on Network and Systems Support for Games (NetGames)*, Oct. 2005.

[42] S. Yang, A. R. Butt, Y. C. Hu, and S. P. Midkiff. Trust but verify: Monitoring remotely executing programs for progress and correctness. In *Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2005.

[43] A. R. Yumerefendi and J. S. Chase. Trust but verify: Accountability for Internet services. In *Proceedings of the ACM SIGOPS European Workshop*, Sep 2004.

[44] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. *ACM Transactions on Storage (TOS)*, 3(3):11, Oct. 2007.

A Correctness

The guarantees in Section 4.7 essentially follow from the properties of the tamper-evident log [22] and the fact that replay is deterministic.

In the following, we use Σ_T to denote the set of observable messages M has transmitted, and Σ_R to denote the set of observable messages M has received. A message is observable if it causally precedes at least one event on a correct node [20]; an example of a non-observable message would be a message that a faulty M secretly sends to another faulty machine, and that is recorded by neither. By *full audit*, we mean an audit of the machine M's entire log, rather than just an individual log segment.

We begin by proving the following lemma, which is another way of stating the log's strong completeness property:

Lemma 1. *If Alice performs a full audit of M, she either learns the sets Σ_T and Σ_R of observable messages that M has sent or received, or she obtains evidence against M that can be verified by a third party.*

Proof. Recall from Section 4.5 that Alice audits M by sending it two authenticators α_i and α_j and challenging it to return the segment $L_{ij} := e_i \dots e_j$ of M's tamper-evident log. A full audit means that α_i and α_j are the authenticators with the lowest and the highest sequence numbers, respectively. Since the authenticators are signed with M's private key, they would convince any third party that the segment $e_i \dots e_j$ must exist. Hence, if M refuses to return the segment, or returns a segment with an invalid hash chain, Alice can use (α_i, α_j) as evidence against M, and a third party receiving this evidence can validate it by challenging M to return L_{ij}, just as Alice did. Thus, the claim holds.

Now suppose that M returns the requested segment, and that its hash chain is intact. From this segment, Alice can extract a set $\bar{\Sigma}_R$ of received messages and a set $\bar{\Sigma}_T$ of transmitted messages. Because all messages are signed and acknowledged, Alice can easily tell if the sets contain a message that was not actually sent or received: If there is a message $m \in \bar{\Sigma}_T \setminus \Sigma_T$ that was recorded but not sent, there cannot be an acknowledgment for m,[3] since M cannot forge the recipient's signature. If there is a message $m \in \bar{\Sigma}_R \setminus \Sigma_R$ that was recorded but not received, m cannot be properly signed for the same reason. If an acknowledgment is missing or L_{ij} contains a message without a proper signature, Alice can use (L_{ij}, α_j) as evidence against M.

Alice can also tell if a message is missing from one of the sets. Because she has been collecting M's authenticators both from messages she received herself (Section 3.3) and from messages and acknowledgments received by other correct nodes (Section 4.6), she can tell if there is an authenticator α_k with $i \leq k \leq j$ that is properly signed by M and nevertheless does not correspond to an entry in L_{ij}. In this case, she can use $(L_{ij}, \alpha_j, \alpha_k)$ as evidence against M. If there is a message $m \in \Sigma_T \setminus \bar{\Sigma}_T$ that M has sent to a correct node but not recorded, the recipient will have collected m's authenticator and forwarded it to Alice. If there is a message $m \in \Sigma_R \setminus \bar{\Sigma}_R$ sent by a correct node that M has acknowledged but not recorded, the recipient will have collected the authenticator from the acknowledgment and given it to Alice as well. Thus, if Alice has not obtained any evidence in this step, and if she has obtained all the authenticators from correct nodes (i.e., no more messages are in flight), she knows that $\bar{\Sigma}_T = \Sigma_T$ and $\bar{\Sigma}_R = \Sigma_R$. Note that we only claim that the sets contain all *observable* messages. □

Theorem 1. *If the machine M is faulty, a full audit of M will report a fault and produce evidence against M that can be verified by a third party.*

Proof. From Lemma 1, we already know that the full audit reveals the sets Σ_T and Σ_R of all observable messages M has sent or received; otherwise Alice obtains evidence against M, and the claim follows trivially.

Now recall that, according to our definition in Section 3.1, M is faulty if and only if it is distinguishable from the reference machine M_R, that is, if M_R cannot produce the same network output when it is started in the same state and is given the same network input. Since Alice has access to a reference copy of the VM image S (Section 4.1), she knows the state in which M started, so she could theoretically decide whether M is faulty by enumerating all possible executions of S in which the messages in Σ_R were received, and by testing whether at least one of them produces the messages in Σ_T. This is not practical in general because for some S the number of candidate executions is infinite.

Crucially, however, M is required to commit to *one particular* execution by including in the log L_{ij} enough information to enable deterministic replay. A correct M can always achieve this by including information about its actual execution; therefore, if 1) Alice cannot replay the log on M_R, or 2) replay succeeds but produces different messages, Alice knows that M must be faulty.

[3]Here we rely on the assumption that Alice and Bob immediately contact each other if they do not receive an acknowledgment from the other side.

Conversely, if M is faulty, it is distinguishable from M_R by definition, so there is *no* execution of M_R that would produce the messages in Σ_T, given the messages in Σ_R. Since Alice's replay on M_R can only produce correct executions, replay must either fail or produce different messages, *no matter what information M includes in the log*. This is the reason why Alice does not need to trust the AVMM (Section 3.4). Furthermore, any third party with access to M_R and (L_{ij}, α_j, S) can repeat Alice's steps and, because replay is deterministic, obtain the same results. Hence, Alice can use (L_{ij}, α_j, S) as evidence against M, and the claim follows. $\square$

Theorem 2. *If the machine M is correct, a full audit of M will not report a fault, and there cannot be any valid evidence against M.*

Proof. If M is correct, then (according to our definition in Section 3.1) there exists an execution of S on M_R that, given the same initial state and the same inputs, produces the same outputs. Hence, M can maintain a linear log that contains these messages, as well as enough information to replay the correct execution. M can respond to Alice's audit request simply by returning the appropriate segment from its log, and it is easy to see that replay will succeed.

To see why there cannot be any valid evidence against M, consider all the possible forms of evidence:

- If the evidence is a properly signed message m that M supposedly did not acknowledge, M can simply accept and acknowledge it when it is challenged with m.

- If the evidence is a challenge to produce a log segment that connects two properly signed authenticators (α_i, α_j), M can respond by returning $e_i \dots e_j$ from its log, since it would not have signed any invalid authenticators, and we have assumed that signatures cannot be forged.

- The evidence cannot be a pair (L_{ij}, α_j) such that α_j authenticates L_{ij} and L_{ij} contains an improperly signed incoming message, because a correct M would have ignored such a message. L_{ij} also cannot contain an unacknowledged outgoing message because in such a case M would have suspected the recipient and Bob would have immediately contacted them.

- The evidence also cannot be (L_{ij}, α_j, S) such that α_j authenticates L_{ij} and replay of L_{ij} starting from S fails on M_R because the only log that could match M's authenticators is M's actual log (hashes are pre-image resistant!) and M has recorded replay information for the correct execution we have assumed to exist.

Therefore, any evidence that is presented against M can either be refuted by M or must be internally inconsistent. Either can be verified by a third party without having to trust Alice or Bob. $\square$