Accountable Virtual Machines

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

Abstract

In this paper, we introduce *accountable virtual machines (AVMs)*. Like ordinary virtual machines, AVMs can execute existing software binaries, but they also 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. Unlike previous solutions, 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, which is based on VMware Workstation, and we have used it to detect several common types of cheating 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 by comparing it to a known-good execution. This capability is useful whenever 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 expect 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 are expected to follow the common protocol and to contribute their fair share of resources;
- customers of a cloud computing service expect the provider to correctly execute their code.

In the above 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 which party 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. In the case of a customer's bug, the provider would like to be able to prove his innocence, and in the case of the faulty provider, 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. This is achieved by running systems inside a virtual machine that (1) maintains a log with enough information to capture 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 runs inside a virtual machine.

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-player games. While AVMs are a general concept that is potentially useful in other contexts, cheat detection is an interesting example application because it is a serious and well-understood problem for which AVMs offer considerable benefits: out of 26 real cheats we downloaded from the Internet, AVMs can detect every single one – without prior knowledge of the nature of the cheats or their implementation.

We have built a prototype AVMM that is based on VMware Workstation, and we have used our prototype to detect several 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 14%, from 156 fps on bare hardware to 134 fps on our prototype, the ping time increases by about 5 ms, and each player must record a log that grows by about 148 MB per hour after compression. Most of this overhead is caused by running VMware Workstation with logging enabled; the additional cost for accountability is comparatively small. The log can be transferred to other players and replayed there if the game is contested.

While our evaluation in this paper focuses on games as an example application, AVMs are a general concept that is potentially useful in other contexts, e.g., to verify that a cloud platform is providing its services correctly and is allocating the promised resources [14]. Our prototype AVMM already supports techniques such as partial audits that would be useful in this context; however, 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 [4, 11, 12]. 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 replay different executions to different auditors.

Recently, a lot of progress has been made on improving the scope and efficiency of replay systems. For example, ODR [1] and SMP-ReVirt [12] offer logging and replay for modern multiprocessor systems, DMP [10] has reduced the logging overhead for multi-threaded programs by making their original execution fully deterministic, and Remus [8] has introduced a highly efficient snapshotting mechanism. AVMMs could directly benefit from each of these improvements.

Accountability: Several systems are available to provide accountability for specific distributed applications, including network storage services [30], peer-to-peer content distribution networks [22], and interdomain routing [2]. Unlike these systems, AVMs are not bound to a particular application. PeerReview [15] is an accountability system that can be applied to arbitrary applications. Our AVMM borrows components from PeerReview – in particular, its tamper-evident log. However, PeerReview requires the target application to be deterministic, and it must be closely integrated with the application, which requires source code modifications and a detailed understanding of the application logic. On the one hand, this potentially results in lower overhead, since knowing the application's semantics enables many useful optimizations. On the other hand, it is infeasible to apply PeerReview to an entire VM image with dozens of applications, and it is nearly impossible if some of the software is only available in binary form. AVMs do not have these limitations; they can make software accountable 'out of the box'.

Remote fault detection: GridCop [29] is a compiler-based technique that can be used to monitor the progress and the correct execution of a remotely executing program by inspecting periodic beacon packets. GridCop is designed for a less hostile environment than AVMs: it assumes that the platform is trusted and that hosts are motivated by self-interest. Also, GridCop does not work for unmodified binaries, and it cannot produce evidence.

A trusted computing platform can be used to detect if a node is running modified software [13, 20]. The approach requires trusted hardware, a trusted OS kernel, and a software and hardware certification infrastructure. Pioneer [26] can achieve a similar property using only software; however, it relies on recognizing sub-millisecond variations of the network round-trip time, which restricts its use to LANs and small networks. AVMs do not require any special hardware support, and they 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 [18]. Several cheat detection techniques are available, such as scanning for known hacks [17, 25] or defenses against specific forms of cheating [5, 23]. In contrast to these, AVMs are generic, that is, they are effective against an entire class of cheats. Chambers et al. [7] uses bit commitment to detect if players lie about their game state. Their approach is similar to the tamper-evident logs used in AVMs; however, the log must be integrated with the game, while AVMs work for existing, unmodified games.

3 Accountable Virtual Machines

3.1 Goals

Figure 1 depicts the basic scenario we are concerned with in this paper (we explain later how to generalize

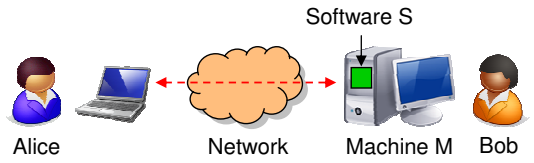


Figure 1: Scenario. Alice is relying on software S, which is running on a machine that is under Bob's control. Alice would like to verify that the machine is working properly, and that Bob has not modified S.

to other scenarios). Alice is relying on Bob to run some software S on a machine M, which is under Bob's control. However, Alice cannot observe M directly, she can only communicate with it over the network. Bob claims that M is working properly, that he has installed the software without modifications and configured it correctly. Our goal is to enable Alice to check whether S, when executing on M, behaves the way she expects.

We assume that there is a reference machine M_R that produces the results Alice expects when executing S. We say that two machines M_1 and M_2 are *indistinguishable* if, given the same initial state and the same sequence of inputs, they produce the same network output. We call the machine M *correct* if it is indistinguishable from the reference machine M_R, provided that M_R is executing the software S. If M is not correct, we say that it is *faulty*. For example, M could be faulty because it is broken, or because Bob has accidentally misconfigured it. Our goal is to provide the following properties:

- **Detection:** If M is faulty, Alice should be able to 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 having to trust either Alice or Bob.

We are particularly interested in solutions that work for any software S that can execute on 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.

3.2 Approach

To detect faults on M, Alice must essentially answer two questions: 1) Which network messages did M send and receive, and 2) is there a correct execution of the software S that is consistent with these messages? The former is easy when M is correct and Bob is honest, but can be challenging otherwise. The latter is difficult because the number of possible executions for any nontrivial software is simply enormous.

Surprisingly, Alice can solve this problem by combining two seemingly unrelated technologies: tamper-evident logs and virtual machines. A *tamper-evident log* [15] 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 cloning or splitting the log.

If Alice has obtained M's message log without detecting any tampering, she must either find a correct execution of S that matches this log, or establish that there isn't one. To help Alice with this, M can additionally record some hints about the execution of S in the log. This 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 extract inputs and nondeterministic events in a generic, application-independent way. Alice then can use deterministic replay [6, 11] to reconstruct a correct execution, provided that one exists.

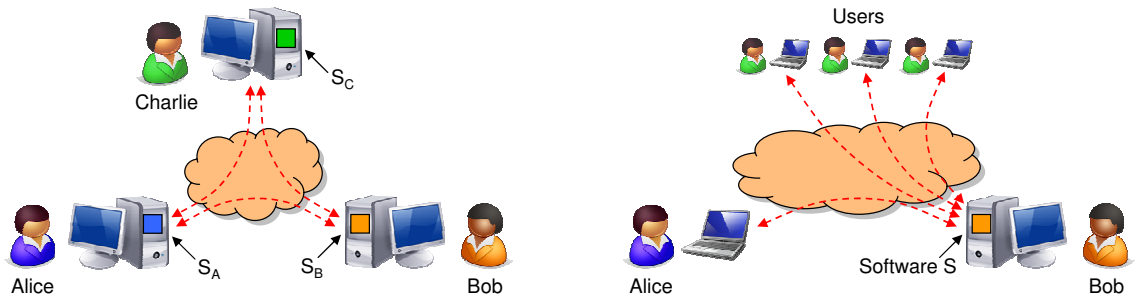
Although the combination of these two technologies may seem obvious in hindsight, we will show that it nevertheless results in a new and powerful capability.

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 the machine M and runs the software S inside an AVM.
2. The AVMM maintains a tamper-evident log of the messages M sends or receives, and it also records hints about the execution of S. When Alice receives a message from M, she detaches the authenticator and saves it for later.
3. 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 for faults.
4. If replay fails or the log cannot be verified against one of the authenticators, Alice can give the software, the log, and the authenticators to a third party, who can repeat Alice's checks and thus verify that a fault has actually occurred.

This generic methodology meets our previously stated goals: Alice can detect faults on M, she can obtain evi-



(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 customers.

dence, 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 is tamper-evident; if he does not, Alice obtains the exact set of observable messages M has sent and received, and since by our definition of a fault there is *no* correct execution of S that is consistent with these messages, deterministic replay inevitably fails on Alice's machine, no matter what the AVMM recorded.

Bob must trust the AVMM to prevent Alice's software from taking over his machine; however, the same would apply to any ordinary VMM. As long as the AVMM correctly runs Alice's software, Alice's can correctly replay the resulting log.

3.5 Should 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. Of course, if Alice chooses to do spot checks, she can only expect to detect faults that manifest themselves in the segments that she actually inspects.

Alice could use various heuristics to choose segments to check. For example, she could inspect the most recent segment when she receives a complaint, she could inspect critical segments where a fault would permanently corrupt the AVM's state, such as key generation phases, or she could simply inspect a random sample of segments. Non-malicious faults, such as hardware faults or misconfigurations, are likely to manifest themselves repeatedly and would be visible in even a small sample. If Bob is planning to misbehave but has a reputation to lose, random sampling would create a risk of detection that may be sufficient to deter him.

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 complicated 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.

In terms of the AVMM algorithm, the only complication that arises from multi-party scenarios is that each machine's authenticators are now distributed over multiple parties, and they must all be collected to ensure that faults can be reliably detected. For example, suppose that in the gaming scenario on the left, Alice wants to audit Charlie's machine. To detect if Charlie has been acting differently towards herself and Bob, she must ask Bob for copies of any authenticators that Charlie's machine may have sent to Bob's. In the scenario on the right, Alice must collect authenticators from the users. If all the users cooperate, Alice can detect all the faults.

If only a subset cooperates, she can still detect all the faults that are observable by that subset [21].

For clarity, we will explain our system mostly in terms of the simple two-party scenario in Figure 1, and we will point out differences to the multi-party scenario where necessary.

4 AVMM design

So far, we have introduced the concept of AVMs. 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 what is the 'correct' behavior.

4.2 Architecture

Our design instantiates each of the building blocks we have described in Section 3.2: a VMM, a tamper-evident log, and a mechanism for deterministic replay. Here, we give a brief overview. Sections 4.4-4.5 describe each building block in more detail.

For the *tamper-evident log*, we adapt a technique from PeerReview [15], which already comes with a proof of correctness [16]. We extend this log to also include the VMM's hints, and we add a mechanism for detecting discrepancies between messages and hints, which could arise, e.g., when a compromised VMM claims to have received different messages than the ones in the log.

The *VMM* we have chosen for this design 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, which can result in a higher overhead.

For *auditing*, 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 the reference software (i.e., the VM image that the machine is expected to use). If the tool finds any discrepancy between the events in the log and the events occurring 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 based on the log used in PeerReview [15]. It 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 $h_i = H(h_{i-1} \,||\, s_i \,||\, t_i \,||\, H(c_i))$ and $h_0 := 0$, 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 $a_i := (s_i, h_{i-1}, h_i, \sigma(s_i \,||\, h_i))$, where the $\sigma(\cdot)$ operator denotes a cryptographic signature with the machine's private key. M also includes h_{i-1} and s_i, so that Alice can recalculate $h_i = H(h_{i-1} \,||\, s_i \,||\, \text{SEND} \,||\, H(m))$ and thus verify that the entry e_i is in fact $\text{SEND}(m)$.

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 above, M's authenticator contains enough information for the recipient to verify that the corresponding entry is $\text{RECV}(m)$. Alice's own acknowledgment contains just a signed hash of the corresponding message, which the

[1] Ensuring deterministic replay on multiprocessor machines requires special care. We will discuss this in Section 7.3.

AVMM logs for the auditor. When an acknowledgment is not received, the original message is retransmitted a few times. If Alice stops receiving messages from Bob's machine altogether, she can ask it to retransmit the last few messages.[2]

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 segment, or if it has forked or cloned its log, 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 [6, 11]. 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 asynchronous 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.

Not all inputs are nondeterministic. For example, the values returned by accesses to the AVM's virtual harddisk need not be recorded. The auditor 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 changing the bytes after the message has been received on M, but 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 disk 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 [8] 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 log, 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 $e_i \ldots e_j$, 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, the auditor will suspect Bob. Alice can also use auditing as a last recourse when messages from the AVM have been

[2] If Bob's machine refuses to respond beyond a certain timeout, Alice can use m as evidence. In practice, she might call Bob on the phone and complain.

lost, including all retransmissions. In this case, Alice can recover the messages from the machine's log.

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 S.

For the syntactic check, the audit tool checks whether all log entries have the proper format, it verifies the cryptographic signatures in each message and acknowledgment, it compares the log against the set of authenticators issued by M, and it checks whether each message has been acknowledged. 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. 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 a 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.

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 two 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, 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 they can jointly decide never to play with Bob again.

4.7 Guarantees

Recall that we have defined a machine M as correct if it is indistinguishable from a reference machine M_R that is running the software M is expected to run, and faulty otherwise (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.

These guarantees hold for any software that can run on M_R. Also, our design makes no assumptions about the nature of the faults that can occur on M (as long as it cannot invert the hash function or break cryptographic keys), and it is robust against malicious behavior by Bob, Alice, or any other user. If spot checking is used, the completeness guarantee extends only to those log segments that are actually audited. A proof of these properties is included in Appendix A.

5 Application: Cheat detection in games

AVMs and AVMMs are general concepts, 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 [25], the Warden [17] or Valve Anti-Cheat (VAC) [27]. These systems work by scanning the user's machine for known cheats [17, 18, 25]; 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

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 random Counterstrike cheats from popular Counterstrike discussion forums

distribute the snapshot to the other players. Each player then initializes his AVM with the agreed-upon snapshot and plays while recording a log to his local disk. After the game, if a player wishes to reassure himself that other players have not cheated, he can request their logs, check them for tampering, and finally replay them using his own, trusted copy of the agreed-upon VM image.

5.3 Which cheats can AVMs detect?

For replay to succeed, the VM images used during recording and replay need to be virtually identical. If different code is executed or different data is read at any time, replay almost certainly fails soon afterward. Because of this, any cheats that need to be installed on the cheater's machine in any way, e.g., as loadable modules, patches, or companion programs, can be detected with an AVM. We hypothesize that this 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 would have to be installed in the AVM with the game to be effective, and would therefore be detected.

Once cheaters know about AVMs, they can try to reengineer their cheats to avoid detection. We found that, in principle, this is possible for 22 of the 26 cheats, e.g., for map hacks and aimbots. In practice, however, implementing such an AVM-aware cheat would be difficult because the cheat could not run anywhere inside the AVM. For example, an aimbot could no longer directly manipulate game state but would instead have to either forge a log (including all the asynchronous events) that has the desired effect but nevertheless produces the same network output during replay, or forge local inputs to aim the player's weapon as desired, e.g., by injecting fake mouse movements. While this is by no means impossible, it raises the bar substantially.

We also found that four of the 26 cheats would be detectable in *any* implementation because they cause the network-visible behavior of the cheater's machine to become inconsistent with any correct execution. This includes cheats such as unlimited ammo, no fall damage, 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.

5.4 Summary

We did not specifically design AVMs for cheat detection, but they do offer three important advantages over current anti-cheating solutions like VAC or PunkBuster: First, they protect players' privacy by separating auditable state (the game in the AVM) from non-auditable state (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?, and
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 [15]. Most of the logging functionality is implemented in a separate process that communicates with the VMM through kernel-level pipes; thus, the AVMM can take advantage of multi-core CPUs by using one of the cores for logging and cryptographic operations and by running AVMs on the other cores at full speed.

Since VMware Workstation only supports uniprocessor replay, our prototype is limited to AVMs with a single virtual core (see Section 7.3 for a discussion of multiprocessor replay). We also implemented and tested support for snapshots and incremental audits, but spot checking offers little benefit for games because a complete audit is practical, so we did not evaluate it in detail.

Our audit tool is implemented as a two-step process: Players first perform the syntactic check using a separate program 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. Because all steps are deterministic, 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, a popular first-person shooter game. 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 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 2.8 GHz Intel Core i7 860 CPUs and 8 GB of memory. 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. One each machine, we installed an AVMM binary that was based on a VMware Workstation OPT 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. We configured the snapshot to disallow software installation; thus, replay fails if a cheater tries to install additional programs or modify existing ones.[3] In the snapshot, the OS is already booted and the player is 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 disabled the frame rate cap in Counterstrike, and we played each game for at least ten minutes.

[3]Otherwise, downloading and installing a cheat would be re-executed during replay without causing any discrepancies.

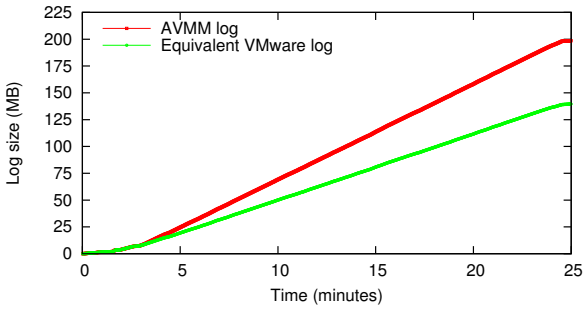


Figure 3: Log growth over time for Counterstrike.

6.3 Functionality check

As a sanity check, we tried out four of the 26 Counterstrike cheats with our prototype. For each cheat, we created a modified VM image that had the cheat preinstalled (recall that the original VM image disallows software installation), 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 cheaters failed due to a divergence during replay.

Examples do not constitute proof, but recall from Section 5.3 that AVMs can detect all of our 26 cheats by design, so this merely validates our implementation.

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 vmware-rec and avmm-rsa768 configurations, and we measured the log size over time. Figure 3 shows the results for the avmm-rsa768 configuration. 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 for each of the three configurations, as well as some details about the content. More than 70% of the log consist of information needed for replay, which in turn consists mainly of `TimeTracker` entries (46%), which are used by the VMM to record the exact timing of events, and MAC-layer events (11%), such as incoming or outgoing net-

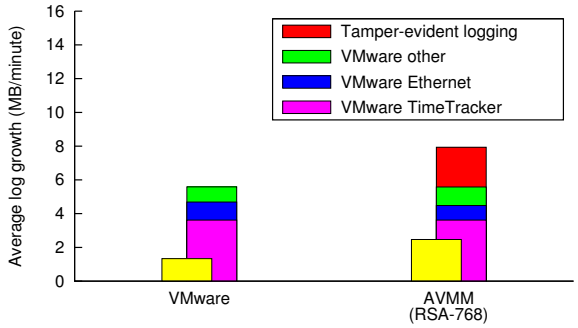


Figure 4: Average log growth for Counterstrike by content. The bars in front show the size after compression.

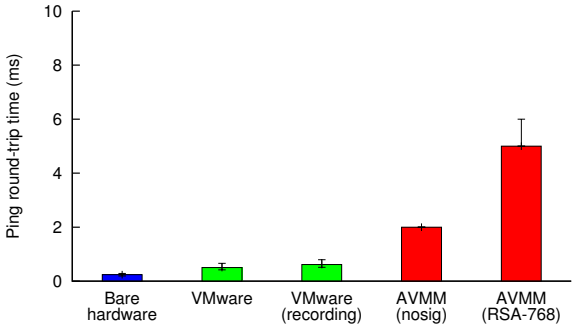


Figure 5: Median ping round-trip times. The error bars show the 5th and the 95th percentile.

work packets. The tamper-evident logging is responsible for the remaining 30%. We also show results after applying `bzip2` and a custom lossless 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, if a player is suspected of cheating, 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. This does not seem problematic because players can leisurely upload their logs to other players in the background after the game has ended. If the game is played over a LAN, e.g., at a competition, the upload would complete in a few seconds.

6.5 Syntactic and semantic check

If a player is suspected of cheating, another player can audit him by checking his log against the authenticators (syntactic check) and by replaying his log using a trusted copy of the VM image (semantic check). We expect the syntactic check to be 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 1,501 seconds) confirm this: We needed 23.2 seconds to compress the log, 8.7 seconds to decompress it, 31.5 seconds for the syntactic check, and 1,370 seconds for the semantic check (1,433 seconds total). Note that replay was actually a bit faster because the AVMM skips any time periods in the recording during which the CPU was idle.

Unlike performance during the actual game, the performance of auditing does not seem overly critical because it can be performed at leisure, e.g., in the background while the machine is used for something else, and it can be omitted entirely if the outcome of the game is not contested. In cases where cheating *is* suspected, it seems like time well spent because it either exposes a cheater or clears an innocent player of suspicion.

6.6 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 34.3 kbps in bare-hw and 220.6 kbps in avmm-rsa768.

This high relative increase is partly due to the fact that Counterstrike sends extremely small packets of 50–60 bytes each, at 25 packets/sec, so the AVMM's fixed per-packet overhead (which includes two cryptographic signatures) has a much higher impact than it would for packets of average size. However, in absolute terms, the traffic is still quite low and well within the capabilities of even a slow broadband upstream. In Section 7.1, we describe an optimization that would considerably reduce this overhead for applications that transfer large objects, such as web servers.

6.7 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 240 μs; adding virtualization increases it to 500 μs, logging to 620 μs, and the daemon to above 2 ms. Enabling 768-bit RSA signa-

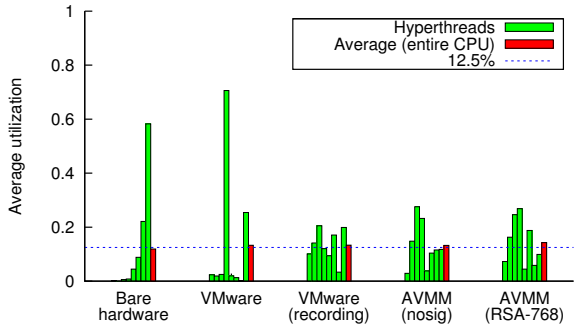


Figure 6: Average CPU utilization in Counterstrike for each of the eight hyperthreads, and for the entire CPU.

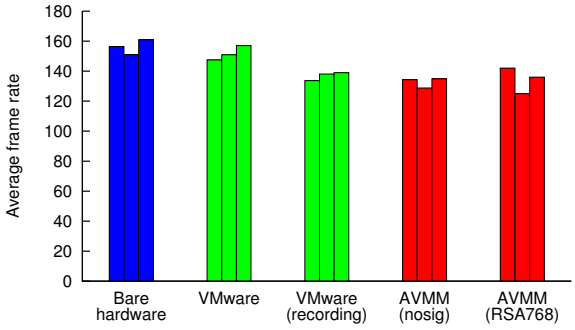


Figure 7: Frame rate in Counterstrike for each of the three machines. The left machine was hosting the game.

tures 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 [9], 5 ms seem acceptable for games. For other applications, the overhead could be reduced by using a more efficient signing algorithm, such as ESIGN [24].

6.8 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 to hyperthread 0 in the AVMM experiments and restricted the game to the other hyperthreads (HTs). Figure 6 shows the average utilization for each HT, as well as the average across the entire CPU. To be conservative, we report numbers for the machine that additionally runs the Counterstrike server.

The low numbers for HT 0 in the AVM experiments (below 8%) show that the overhead from the tamper-evident log is low. We also see that the game is constantly busy rendering frames. Consider that the Counterstrike rendering engine is single-threaded, so it cannot run on more than one HT at a time. Because the OS will sometimes schedule it on one HT and sometimes on another, we expect to see an average utilization of 12.5% on the eight HTs, which our results confirm.

6.9 Frame rate

Since the game is constantly rendering frames, a more 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 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 duration of the game. Figure 7 shows our results for each of the three machines. The results vary 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 14% 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 10%. In absolute terms, the resulting frame rate (134 fps) is still very high; posts in Counterstrike forums generally recommend configuring the game for about 40–60 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 8 fps.

6.10 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 14%, compared to the rate seen on bare hardware. In return for this cost, players gain the ability to audit other players if they suspect them of having cheated. 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, accountability and virtualization.

7 Discussion

7.1 Other applications

In this paper, we have described the basic concept of AVMs, and we have evaluated them in the context of

multi-player games. However, AVMs are application-independent, and with some straightforward optimizations they could potentially be used in other contexts. We have developed a set of extensions that could be used to adapt AVMs for cloud computing [28], but here we can only summarize them briefly due to lack of space.

In the cloud, AVMs face three main challenges: 1) auditors cannot possibly replay the *entire* execution; 2) accountable services must be able to interact with non-accountable clients; and 3) it is not practical to sign every single packet. The first challenge can be addressed with spot checking (Section 3.5). To address the second, we have developed two *proxies* that can mediate between AVMs and legacy software by transparently adding and removing authenticators as needed. One proxy is a standalone application that interposes on TCP and UDP connections; this can be used with many standard tools, such as `ssh`. The other is a combination of JavaScript and a Java applet that is specifically optimized for cloud-based web services.

To address the third challenge, we have added an *aggregator* to the AVMM that extracts larger objects from TCP connections. The aggregator works like a middlebox, much like the proxies used by many cellular networks. For example, when a client requests a large image from an AVM-enabled web server, the aggregator in the AVMM interposes on the TCP connection, locally downloads the entire image, attaches a single authenticator to it, and only then forwards the image to the client. This optimization slightly changes the TCP semantics, but it dramatically reduces the CPU and network overhead in this setting.

7.2 Using trust to get stronger guarantees

One of the strengths of AVMs is that they do not require any trusted components to verify a remote node's execution. However, if we extend the technique to include such trusted components, we can obtain additional guarantees. The two extensions we discuss are secure local input and trusted AVMMs.

Secure local input: AVMs cannot detect the hypothetical re-engineered aimbot from Section 5.3 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 be build.

Trusted AVMM: If we can trust the AVMM that is running on a remote node, we can also prevent information leakage attacks, such as map hacks or transparent walls. 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 it could ensure that no information leaks occur through other channels, for example by preventing outside access to the graphics card. If the trusted platform on Bob's machine includes trusted hardware, it may also prevent Bob from reading the information directly from memory. Lastly, remote attestation could be used to make sure that a trusted AVMM is indeed running on a remote computer.

7.3 Replay for multiprocessors

We have not evaluated our prototype AVMM with multiprocessor VMs because the VMM we have chosen supports only deterministic replay for uniprocessors. SMP-ReVirt [12] has recently demonstrated that replay is also possible for multiprocessors, but its cost is substantially higher than the results we have shown here. However, replay is a building block for many useful applications, such as forensics [11], replication [8], and debugging [19]. Therefore, we believe that there is considerable interest in developing more efficient techniques. In fact, some important advances have already been made [1, 10]. As more efficient techniques become available, AVMMs can directly benefit from them.

8 Conclusion and future work

In this paper, we have introduced accountable virtual machines (AVM), which 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 that the machine has been working correctly.

To demonstrate that AVMs are feasible, we have designed and implemented an AVMM that provides AVMs on commodity PCs. We have applied AVMs to detect several common forms of cheating in Counterstrike, a popular online multi-player game, and our results show that they are practical in this scenario. However, AVMs are in no way specific to games; they have interesting applications in other domains.

Interesting opportunities for future work include the development of AVMMs that record less information than needed for deterministic replay, e.g., using the approach from ODR [1], or detection methods that do not require full replay, e.g., by skipping over computations that do not contribute to network-level outputs.

References

[1] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 193–206, October 2009.

[2] David Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Accountable Internet protocol (AIP). In *Proceedings of the ACM SIGCOMM Conference*, pages 339–350, August 2008.

[3] Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmerman. Factorization of a 768-bit RSA modulus. `http://eprint.iacr.org/2010/006.pdf`.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, October 2003.

[5] Nathaniel E. Baughman, Marc Liberatore, and Brian Neil Levine. Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Transactions on Networking*, 15(1):1–13, February 2007.

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

[7] Chris Chambers, Wu-Chang Feng, Wu-chi Feng, and Debanjan Saha. Mitigating information exposure to cheaters in real-time strategy games. In *Proceedings of the international workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 7–12, June 2005.

[8] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, pages 161–174, April 2008.

[9] James Dabrowski and Ethan V. Munson. Is 100 milliseconds too fast? In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*, pages 317–318, April 2001.

[10] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, March 2009.

[11] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[12] George W. Dunlap, Dominic Lucchetti, Peter M. Chen, and Michael Fetterman. Execution replay for multiprocessor virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, March 2008.

[13] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 193–206, October 2003.

[14] Andreas Haeberlen. A case for the accountable cloud. In *Proceedings of the 3rd ACM SIGOPS International Workshop on Large-Scale Distributed Systems and Middleware (LADIS'09)*, October 2009.

[15] Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 175–188, October 2007.

[16] Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. Technical Report 2007-3, Max Planck Institute for Software Systems, October 2007.

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

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

[19] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, Apr 2005.

[20] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, April 2009.

[21] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Sasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[22] Nikolaos Michalakis, Robert Soulé, and Robert Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *Proceedings of the 4th*

USENIX Symposium on Networked Systems Design & Implementation (NSDI), April 2007.

[23] Christian Mönch, Gisle Grimen, and Roger Midtstraum. Protecting online games against cheating. In *Proceediongs of the 5th Workshop on Network and System Support for Games (NetGames)*, page 20, October 2006.

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

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

[26] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

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

[28] Alexander Wieder. Accountable web applications. Master's thesis, Max Planck Institute for Software Systems (MPI-SWS), 2009.

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

[30] Aydan R. Yumerefendi and Jeffrey S. Chase. Strong accountability for network storage. *ACM Transactions on Storage*, 3(3):11, 2007.

A Correctness

The guarantees in Section 4.7 essentially follow from the properties of the tamper-evident log [16] and the fact that replay is deterministic. 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 complete and accurate set of observable messages 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 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 S_R of received messages and a set S_T of transmitted messages. Because all messages are signed and acknowledged, Alice can easily tell if the sets are not accurate, that is, if they contain a message that was not actually sent or received: If a message was not sent, there cannot be an acknowledgment,[4] since M cannot forge the recipient's signature; if a message was not received, it cannot be properly signed for the same reason. If either of these applies nevertheless, Alice can use (L_{ij}, α_j) as evidence against M.

Alice can also tell if the sets are not complete. Because she has been collecting M's authenticators both from messages she received herself (Section 3.3) and from messages 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. Note that we only claim that the sets contain all *observable* messages, that is, messages that either directly or transitively affect a correct node. An example of a non-observable message is a message m that M secretly sends □

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 complete and accurate sets S_T and S_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 would produce different network output given the same network input and starting in the same state. Since Alice has access to a reference copy of the VM image V (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 V in which the messages in S_R were received, and by testing whether at least one of them produces the messages in S_T. This

[4] 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.

is not practical in general because for some V 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. Conversely, if M is faulty, it is distinguishable from M_R by definition, so there is *no* correct execution of M_R that would produce the messages in S_T, given the messages in S_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, V) can repeat Alice's steps and, because replay is deterministic, obtain the same results. Hence, Alice can use (L_{ij}, α_j, V) as evidence against M, and the claim follows. □

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 is an execution of V 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 $e_i \ldots e_j$ 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, V) such that α_j authenticates L_{ij} and replay of L_{ij} starting from V 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. □