# Trusted VM Snapshots in Untrusted Cloud Infrastructures

Abhinav Srivastava[1], Himanshu Raj[2], Jonathon Giffin[3], and Paul England[2]

[1] AT&T Labs–Research
[2] Microsoft Research
[3] School of Computer Science, Georgia Institute of Technology
abhinav@research.att.com, {rhim,pengland}@microsoft.com, giffin@cc.gatech.edu

**Abstract.** A cloud customer's inability to verifiably trust an infrastructure provider with the security of its data inhibits adoption of cloud computing. Customers could establish trust with secure runtime integrity measurements of their virtual machines (VMs). The runtime state of a VM, captured via a snapshot, is used for integrity measurement, migration, malware detection, correctness validation, and other purposes. However, commodity virtualized environments operate the snapshot service from a privileged VM. In public cloud environments, a compromised privileged VM or its potentially malicious administrators can easily subvert the integrity of a customer VMs snapshot. To this end, we present *HyperShot*, a hypervisor-based system that captures VM snapshots whose integrity cannot be compromised by a rogue privileged VM or its administrators. HyperShot additionally generates trusted snapshots of the privileged VM itself, thus contributing to the increased security and trustworthiness of the entire cloud infrastructure.

## 1 Introduction

The lack of verifiable trust between cloud customers and infrastructure providers is a basic security deficiency of cloud computing [2, 14, 25, 31]. Customers relinquish control over their code, data, and computation when they move from a self-hosted environment to the cloud. Recent research has proposed various techniques to protect customers' resources in the cloud [10, 19, 34]. We consider an alternate way to establish trust and provide customers with control: runtime verification of their rented virtual machines' (VMs) integrity. The runtime state of a VM, captured via a *snapshot*, can be used for runtime integrity measurement [3, 5, 12], forensic analysis [42], migration [8], and debugging [39]. The snapshot allows customers to know the state of their VMs and establishes trust in the cloud environment.

Today's commodity virtualization environments such as Xen [4], VMware [41], and Hyper-V [24] capture a consistent runtime state of a virtual machine at a point in time via the *snapshot* service. Each of these virtualized environments generates a snapshot from a privileged VM, such as dom0 [4] or the root VM [24]. Since the privileged VM, together with the hypervisor and the hardware, comprises the infrastructure platform's trusted computing base (TCB), the snapshot generation service and the resulting snapshot stored in the privileged VM are inherently trusted.

Such unconditional trust in privileged VMs is sensible for a self-hosted private virtualized infrastructure, but it does not generalize to today's virtualization-based public cloud computing platforms due to different administrative control and restrictions [6]. Cloud customers or tenants must trust the root VM to generate a true snapshot of their VMs. Given that the root VM runs a full fledged operating system and a set of user-level tools with elevated privileges, any vulnerability present in that substantial software collection may be exploited by attackers or malware to compromise the integrity of the snapshot process or the snapshot information itself. Customers must hence also rely on an infrastructure provider's administrators and administration policies to properly manage the security of privileged VMs. Yet, the problem of malicious administrators is serious enough that researchers have proposed technical measures to support multiple-person control for administration [30].

In this work, we focus on the trust issues between customers and providers and propose a system called *HyperShot* that generates trusted snapshots of VMs *even in the presence of adversarial root VMs and administrators*. HyperShot only relies on a hypervisor and does not include the root VM in its TCB. Trust in the hypervisor itself is established via hypervisor launch using Trusted Execution Technology (TXT) [15]. Our design departs from the existing snapshot generation approaches in that HyperShot operates from the hypervisor rather than from the root VM. Since the hypervisor executes at a higher privilege level than the root VM, this design protects the snapshot service from the compromised root VM. HyperShot protects the integrity of the generated snapshot with a cryptographic signature from a Trusted Platform Module (TPM) that incorporates measurements of all trusted components and a hash of the snapshot itself. Since the TPM is exclusively under the control of our hypervisor, these measurements enable a verifying entity or customer to establish trust in a VM's snapshot. Since HyperShot does not trust the root VM and its administrators, it extends the same snapshot generation functionality and trust guarantees to the root VM itself.

To allow customers and providers to obtain verifiable snapshots of VMs executing in the cloud, we present a snapshot protocol that operates with *minimal trust in the cloud infrastructure itself.* The design of HyperShot decouples the snapshot generation process from the verification or analysis process. A customer can generate the snapshot in the cloud and can perform analysis at its own end or assign the verification duties to a third party. This design reduces burden on cloud infrastructure providers since customers can choose analysis software independent of the cloud provider. HyperShot's snapshot of the root VM enables customers to verify the integrity of a provider's management VMs in a measurable way with the help of third parties trusted by both providers and customers. We believe that this significantly contributes to the trustworthiness and reliability of the cloud infrastructure provider as a whole.

To demonstrate the feasibility of our ideas, we have implemented a HyperShot prototype based on Microsoft's Hyper-V virtualization infrastructure; any other hypervisor such as Xen or VMware is equally suitable for our work. The platform TCB in commodity virtualized environments is large and contains substantially more code than just the hypervisor due to the inclusion of one or more privileged VMs to perform tasks like I/O virtualization, peripheral access, and management [37]. Our design significantly reduces the platform TCB for the snapshot service by removing the privileged VM and its administrators from the TCB, an approach that is similar to the VMware ESXi architecture [40]. HyperShot only adds $\sim 4K$ lines to the hypervisor codebase, which coupled with the eviction of privileged VMs, nets a much smaller TCB.

Our choice of a commodity virtualization environment is motivated by the reliance of today's cloud infrastructures on full featured hypervisors, rather than on small security-specific hypervisors like SecVisor [36] and TrustVisor [20]. These small hypervisors do not support many features essential to public cloud environments, such as support of multiple VMs and VM migration. Nevertheless, our trusted snapshot service is general, and it can easily support security-specific hypervisors if cloud providers decide to move towards these smaller hypervisors in the future.

To demonstrate the usefulness of trusted snapshots, we integrated HyperShot with the open source forensic utility *Volatility* [42] and with runtime integrity measurement software called *KOP* [5]. Our security evaluation demonstrates HyperShot's effectiveness at mitigating various attacks on the snapshot service and on the snapshot itself. Our detailed performance evaluation shows that HyperShot incurs a modest runtime overhead on a variety of benchmarks.

In summary, we make the following contributions:

- We investigate the trust issues between cloud customers and providers and motivate the need to reduce the effective platform TCB in the cloud by showing a concrete attack on the snapshot originating from a malicious root VM.
- We propose trusted snapshots as a new cloud service for customers and design a mechanism for trusted snapshot generation across all VMs, including the root VM. Our design mitigates various attacks that compromise the snapshot service and integrity of the generated snapshot. We implemented our design in the Hyper-V hypervisor.
- We associate a hardware rooted trust chain with the generated snapshot by leveraging trusted computing technologies. In particular, the hypervisor *signs* the snapshot using a TPM. Trusted boot using TXT establishes trust in the hypervisor.
- We present a snapshot protocol allowing clients to request and verify VM snapshots. We demonstrate the use of trusted snapshots in various end-to-end scenarios by integrating the trusted snapshot with forensic analysis and runtime integrity measurement software.

## 2 Overview

### 2.1 Threat Model

Our threat model considers attacks that compromise the integrity of the snapshot either by tampering with the snapshot file's contents or with the snapshot service. We refer to all entities that could perpetrate this attack from the root VM—whether it is a malware instance running in the root VM or a malicious administrator—collectively as a *malicious root VM*. In particular, we consider the following types of attacks:

- **Tampering:** A malicious root VM may modify the generated snapshot file or the runtime memory and CPU state of a target VM during the snapshot process to

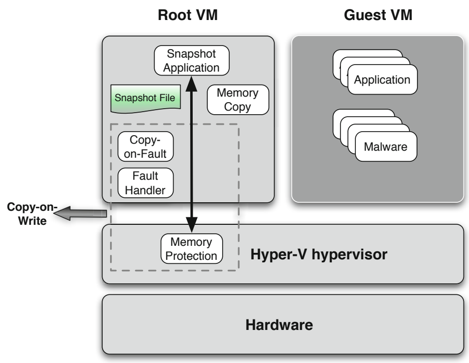

**Fig. 1.** The existing Hyper-V snapshot design places all components other than memory protection inside the root VM, which makes the snapshot service vulnerable to attacks by the root VM

remove memory regions that include evidence of malware or other activity undesired by the customer.
- **Reordering:** A malicious root VM may reorder the content of memory pages in the snapshot file without modifying the contents of individual pages. This may fool integrity checking or forensic analysis utilities that expect a snapshot file's contents to be in certain order, leading to failures to locate security-relevant data in the file.
- **Replaying:** A malicious root VM may provide an old snapshot of a target VM that does not contain any malicious components.
- **Masquerading:** A malicious root VM may try to intercept the snapshot request and modify the request's parameters, such as the target VM, to provide a snapshot of a VM different than the one intended.

Note that attacks meant to compromise the snapshot process via introduction of a bad hypervisor, such as bluepill [32] and SubVirt [17], are addressed by the use of trusted computing techniques, as described later. We do not consider hardware attacks, such as cold boot attacks on RAM and side channel attacks, specifically since these attacks may compromise the confidentiality of a VM's state while HyperShot's goal is to protect the integrity of the snapshot. Finally, we do not consider availability attacks, such as deleting the snapshot file, crashing the root VM, or crashing the customer VMs during the snapshot, and DMA-based attacks from rogue devices or rogue device drivers. DMA-based attacks can be thwarted using IOMMU-based protection methods, as successfully demonstrated by Nova [37] and TrustVisor [20]. We also assume *minimal trust in the cloud infrastructure* to assign a globally unique identifier (guid) to a customer VM and enforce the invariant that a VM with a particular guid can only be executing on one physical machine at any given time. This infrastructure is independent of customers, maintained by cloud providers, and it can be achieved with a much restricted core set of machines. This assumption is already required in today's cloud environments such as EC2 and Azure for proper functioning of network-based services.

## 2.2 Threats to Existing Hyper-V Snapshot Mechanisms

A typical virtualization infrastructure includes a hypervisor, multiple guest VMs, and a privileged management VM, such as the root VM or dom0. The current virtualization architecture supported by Hyper-V, Xen, and VMware ESX server allows the snapshot service to operate from the privileged management VM. As shown in Figure 1, the root VM in Hyper-V takes a guest VM's snapshot with only minimal support from the hypervisor. More specifically, the root VM only relies on the hypervisor to protect guest memory pages from writes performed by the target guest VM being snapshotted. These writes trigger the root VM's copy-on-write (CoW) mechanism, where the root VM handles faults (using a fault handler), copies the content of the page (using copy-on-fault) before removing the protection, and resumes the guest VM's execution. Concurrently, the snapshot application also copies other guest memory pages. After completion of the snapshot, the snapshot file is stored in the root VM and CoW protection on guest memory pages is removed.

We evaluated the security of the existing Microsoft Hyper-V [24] snapshot mechanism in a cloud environment under the threat model described above. We developed a concrete tampering attack on a customer's snapshot file by removing from it evidence of malware infection and other important information that a malicious administrator may want to hide from a customer. To launch the attack, we utilized a forensic analysis utility called *Volatility* to extract information such as the list of running processes, loaded drivers, opened files, and connections. We first opened the snapshot file in analysis mode and listed all running processes, and we then chose a process from the list to remove—in a real threat scenario, this could be malware. Next, we used Volatility to alter the list of running processes by rewriting the linked list used by Windows to store all running processes. We repeated this experiment to remove a loaded driver from the list of drivers. These malicious modifications will not be detected by the consumers of this snapshot due to the lack of any measurable trust associated with the generated snapshot.

## 3 HyperShot Architecture

In this section, we present the design goals of HyperShot and the detailed description of its various components as shown in Figure 2.

### 3.1 Design Goals

We designed HyperShot to satisfy the following goals:

- **Security:** HyperShot creates trusted snapshots by protecting both the snapshotting code and the generated snapshot data files against manipulation by malicious root VMs. To secure the snapshot service, HyperShot deploys its components inside the hypervisor. Since the hypervisor runs in a high-privileged mode, the untrusted root VM cannot alter HyperShot's components either in memory or in persistent storage. To preserve the integrity of the snapshot files stored in the root VM, HyperShot hashes the snapshot data of the target VM[1] from the hypervisor and signs the hash using the TPM. While manipulation is not directly prevented, the signed hashes

[1] We use the term target VM to refer to any VM that is being snapshotted without distinguishing whether it is a guest VM or the root VM. Any distinction is explicitly qualified.

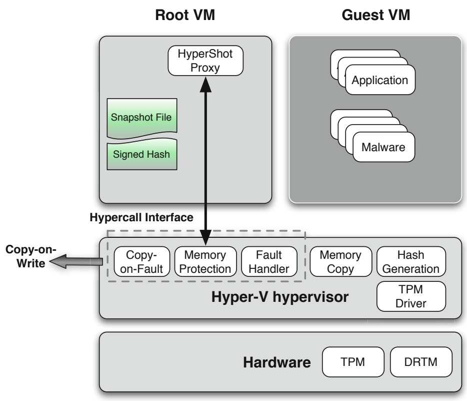

**Fig. 2.** HyperShot's design contains all components inside the hypervisor. It runs only a thin proxy in the root VM to forward snapshot requests to the hypervisor. The generated snapshot and its signature are stored in the root VM.

allow HyperShot or a customer to detect malicious alterations by the infrastructure provider. Unlike existing snapshot generation techniques that include the root VM in the platform TCB, HyperShot excludes the root VM and its administrators from the TCB; hence it extends the same snapshot generation functionality and trust guarantees to the root VM.

– **Consistency:** To capture a consistent state of the target VM, HyperShot does not allow any modification to the target VM state without first recording the state as it was at the time of the snapshot request.
– **Performance:** To keep the performance overhead moderate, HyperShot only pauses the target VM for a minimal duration and then uses copy-on-write (CoW) to allow the VM to continue execution during the snapshot. This design incurs low overhead on applications executing inside the target VM.

HyperShot deploys all components inside the hypervisor; it keeps only a thin proxy client in the root VM. HyperShot also protects the integrity of the snapshot from attacks that originate due to a compromise of the proxy software executing inside the root VM, as explained in Section 5.1. HyperShot moves only snapshot service related functionality into the hypervisor, leaving the rest of the code inside the root VM. The additional code added into the hypervisor totals $\sim 4K$ lines of source.

## 3.2 Enhanced Copy-on-Write Protection

HyperShot creates trusted and consistent snapshots of VMs executing in a virtualization based cloud environment. One possible approach to enable consistency is to pause the target VM during the entire snapshot process. This approach may impose severe runtime overhead on applications running in the target VM during the snapshot generation. Further, it still does not guarantee consistency since a malicious root VM may modify the state of the target VM while the target VM's snapshot is in progress. *To offer both consistency and security*, HyperShot utilizes an enhanced copy-on-write (CoW) mechanism inside the hypervisor.

To set up the enhanced CoW on a guest VM at the beginning of a snapshot, the hypervisor pauses the guest VM and marks its memory pages read-only by iterating across its address space. To protect the guest VM's state from untrusted modifications by the root VM during the snapshot, HyperShot also write-protects the corresponding memory pages mapped in the root VM's page tables. For snapshots of the root VM, the CoW setup is performed only on the root VM's address space because guest VMs cannot access the root VM's memory pages. Our CoW mechanism is different from those used in live VM migration [8]. In particular, we copy memory page content prior to modification rather than iteratively copying dirty pages after alteration.

## 3.3 Access Mediation

CoW setup allows HyperShot to mediate writes performed by the target VM to write-protected memory pages. This design fulfills HyperShot's goals of *security and consistency* since it allows HyperShot to copy the content of memory pages before they are modified. In particular, at each page fault on a write-protected page, HyperShot copies the content of the faulted page before restoring the original access permissions. These copies are stored in protected memory regions belonging to the hypervisor. In our implementation, this memory is *deposited* by the root VM on-demand before the snapshot of a VM is started. Once deposited, this memory becomes inaccessible from the root VM, hence the snapshot data stored in this memory cannot be modified by the malicious root VM. HyperShot keeps the content of the faulted and copied pages inside the hypervisor until the snapshot process is completed. *This design is required both for security and correctness* because HyperShot allows changes to occur on the faulted pages after copying the pages' contents. Multiple guest physical addresses (GPAs) may map to the same system physical address (SPA), so if a write-protected GPA faults and it mapped to an SPA whose page has already been processed, then HyperShot takes the snapshot of the faulted GPA and its hash from the stored copy rather than directly from the target VM's memory as the memory content may have been modified.

For the root VM's snapshot, faults originate only from the root VM. However, for a guest VM's snapshot, HyperShot receives faults on a write-protected page both from the guest and root VM. The faults occur from the guest VM as part of its own execution and from the root VM as part of I/O and other privileged operations. The stock Hyper-V hypervisor does not expect the root VM to generate page faults due to page protection bits because the root VM has full access to all guest VMs' memory pages. To handle these new faults, HyperShot adds a new page fault handler in the hypervisor to facilitate the copying of page contents and restoring of the original access permissions in the root VM's address space. The same page fault handler is used during the snapshot of the root VM to handle page faults due to CoW setup on the root's address space.

Finally, HyperShot provides persistent protection to the runtime environment of the target VM. HyperShot mediates operations that map and unmap memory pages in the target VM's address space maintained by the hypervisor. If these changes involve a page that is protected and not already copied, HyperShot copies the contents of the memory page before it allows the operation to complete. In addition to recording memory pages, HyperShot also snapshots the virtual CPU (vCPU) state associated with a target VM at the beginning of a snapshot and stores all vCPU register values inside the hypervisor.

## 3.4 Memory Copy

The memory copier is a component that resides in the hypervisor and closely works with the HyperShot proxy to move memory contents into a generated snapshot file. The proxy invokes a series of hypercalls in a loop during which it sequentially requests the contents of each memory page by invoking the memory copier and writes to the file. The copier is responsible for two tasks. First, it passes the contents of pages already snapshotted via CoW to the proxy. Second, it snapshots remaining memory pages that were not modified during the CoW, as the CoW mechanism would not have recorded the contents of unwritten pages.

On a request from the HyperShot proxy, if the requested page is write-protected and not already copied, the copier extracts the contents of the page from the target VM's memory, calculates a hash over the page's content, and stores only its hash in the hypervisor for the future use. It passes a copy of the content back to the HyperShot proxy. If the requested page had already been copied during the CoW, the copier calculates the hash on the previously stored content inside the hypervisor and sends the content back to the proxy. Finally, if a requested memory page is not mapped in the target VM, all zeroes are returned and a hash of the zero page is stored for the future use.

## 3.5 Hash Generation

To protect the integrity of the snapshot file from a malicious root VM, HyperShot creates message digests or hashes of the target VM's memory pages before the snapshot content is sent to the root VM. These hashes are stored inside the hypervisor, and thus are not accessible to the root VM.

The hashing process works in two steps. In the first step, HyperShot generates the SHA-1 hash of each individual memory page present in the target VM's address space. In the second step, it creates the composite hash of all the individual hashes calculated in the first step:

$$\mathtt{H_{composite}} = \mathtt{SHA\text{-}1}(\mathtt{H_1}||\mathtt{H_2}||\mathtt{H_3}......||\mathtt{H_M})$$

where M is the total number of guest memory pages and $\mathtt{H_i}$ is the SHA-1 hash of the $i^{th}$ memory page.

To generate the composite hash, HyperShot follows a simple ordering: it always concatenates individual hashes starting from the hash of the first memory page in the target VM and continues up to the hash of the last page. This ordering is important—as described earlier in Section 2.1, an attacker may launch a reordering attack by altering the snapshot file. To detect the page reordering attempts, HyperShot always expects the snapshot to be performed in sequential order, and it generates the composite hash in the similar fashion. This design detects a page ordering attack as the composite hash will not match during verification. We describe the snapshot verification process in Section 4.2. We used a linear hash concatenation approach for simplicity, though other efficient techniques, such as Merkle hash trees [22], could be used to generate a composite.

## 3.6 Integrity Protection of the Snapshot

HyperShot keeps the contents of CoW memory pages and the hashes of all pages in the hypervisor memory until the snapshot is over; the individual hashes are used to generate the composite. Merely generating $\mathtt{H}_{\mathtt{composite}}$ inside the hypervisor is insufficient to maintain the integrity of the snapshot because malware or malicious administrators can easily subvert the hash when it is transferred to the root VM. We must protect the integrity of $\mathtt{H}_{\mathtt{composite}}$ itself so that it cannot be modified inside the root VM. We use hardware-supported *signing* in order to detect malicious tampering of $\mathtt{H}_{\mathtt{composite}}$ and leverage trusted computing technologies to establish a trusted environment for the signing operation.

Trusted computing technologies provide a basis for trusting a platform's software and hardware configurations based on a *trust chain that is rooted in hardware*. In particular, it provides the *Trusted Platform Module* (TPM) [38], which is available today on most commodity hardware platforms. Each TPM has multiple 20-byte *Platform Configuration Registers* (PCRs) used to record measurements related to the platform state, such as those of various software components on the platform. These measurements form a chain of trust, formed either *statically* at the platform's boot time, or *dynamically* at any time during execution. Most PCRs cannot be reset by software but only *extended* from their previous values. The TPM also contains an asymmetric *endorsement key* (EK) that uniquely identifies the TPM and the physical host it is on. Due to security and privacy concerns, a separate asymmetric *attestation identity key* (AIK) is used as an alias for EK when signing *remote attestations* of the platform state (as recorded in various PCRs). The process of establishing an AIK based on EK is described in detail in [27]. The remote attestation process is based on the TPM's *quote* functionality, which signs a 20-byte sized data value using $\mathtt{AIK}_{\mathtt{priv}}$ and a set of PCRs [38]. A more detailed description of trusted computing technologies for today's commodity platforms is provided by Parno et al. [28].

HyperShot leverages a research prototype version of Hyper-V that provides a trusted boot of the hypervisor. In particular, the hypervisor is launched via *Intel's TXT* [15] technology, and it is started *before* the root VM (compared to the stock Hyper-V architecture where the hypervisor starts after the root VM). This late hypervisor launch architecture is very similar to Flicker [21] and trusted Xen [7]. The trusted boot measures the platform TCB and records the measurements in a non-repudiable fashion in the TPM's PCRs 17, 18, and 22. The AIK is also loaded into the TPM before a signature can be requested from the TPM in the form of a quote.

After the snapshot process is completed, the hypervisor records $\mathtt{H}_{\mathtt{composite}}$ as part of the system configuration in the resettable PCR 23. The quote request (described in detail in Section 4) includes PCRs 17, 18, 22, and 23 in the signing process to ensure (a) the integrity of the platform TCB via PCRs 17, 18, and 22; and (b) the integrity of the generated snapshot itself via PCR 23. Note that in HyperShot, the TPM is under the control of hypervisor and only a para-virtualized interface is provided to the root VM to request a quote. A malicious root VM cannot send arbitrary raw TPM commands to corrupt the TPM state. Also, the para-virtualized TPM interface presented to the root VM does not allow resetting of PCR 23; only hypervisor is allowed to reset PCR 23. This design protects the integrity of the measurement from the malicious root VM.

### 3.7 DMA Considerations

A DMA operation occurring during a snapshot may alter memory pages of a target VM. To ensure snapshot consistency, we modify the guest DMA processing code inside the root VM at a point prior to the actual DMA. We add a hypercall in this code path to pass the GPAs of memory pages being locked for the DMA. This ensures that the original memory page is processed prior to the DMA if it had not already been copied for the snapshot. We intercept guest DMA operations for consistency, and not for security reasons. Our DMA interception code resides inside the root VM and is prone to attacks. Our current implementation does not yet support interception of the root VM's DMA operations. Though it is possible by modifying the root's DMA handler, it would also not be secure. A future integration of IOMMU with CoW based protection will ensure that any DMA write operation that bypasses the snapshot will fault and be either aborted or restarted after the copy is made.

## 4 HyperShot Protocol

The snapshot protocol allows clients or challenging parties to securely retrieve trusted snapshot and validation information from the providers or attesting system. HyperShot generates snapshots of VMs at the request of a client. The protocol used by HyperShot involves the following entities:

- **Challenger:** A customer or challenger is an entity who wishes to measure the integrity of her VM running in the cloud. In HyperShot, a cloud infrastructure provider can act as the challenger when it requests snapshots of the root VM of a particular machine. Challengers can act as verifiers or can assign verification duty to third parties whom they trust.
- **HyperShot Service Front-End:** The front-end routes the snapshot request to the physical machine where the VM to be snapshotted is currently executing. It also sends the snapshot file and TPM quote along with the `AIK` certificate back to the customer. The HyperShot service front-end is a part of the minimal trusted cloud infrastructure.
- **Forwarder:** The HyperShot proxy receives the request from the HyperShot service front-end and uses the hypercall interface to pass the request to the hypervisor.
- **Generator/Attestant:** The generator is the HyperShot component inside the hypervisor that performs most of the work—it sets up proper CoW protections, copies memory pages, protects the integrity of individual memory pages and the composite snapshot, and provides an attestation using the TPM. This is the key component of the HyperShot TCB.

### 4.1 Snapshot Generation Protocol

Figure 3 depicts the HyperShot protocol with steps involved in the snapshot generation process. The protocol starts when a challenger sends a snapshot request to the HyperShot service front-end in the cloud, identifying the VM to be snapshotted by the guid

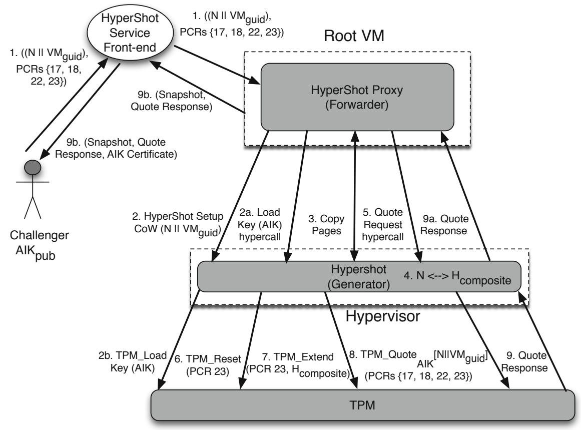
HyperShot Service Front-end
1. ((N ll VMguid), PCRs {17, 18, 22, 23})
Root VM
1. ((N ll VMguid), PCRs {17, 18, 22, 23})
HyperShot Proxy (Forwarder)
9b. (Snapshot, Quote Response)
9b. (Snapshot, Quote Response, AIK Certificate)
Challenger AIKpub
2. HyperShot Setup CoW (N ll VMguid)
2a. Load Key (AIK) hypercall
3. Copy Pages
5. Quote Request hypercall
9a. Quote Response
Hypershot (Generator)
4. N <--> Hcomposite
Hypervisor
2b. TPM_Load Key (AIK)
6. TPM_Reset (PCR 23)
7. TPM_Extend (PCR 23, Hcomposite)
8. TPM_Quote AIK [NllVMguid] (PCRs {17, 18, 22, 23})
9. Quote Response
TPM

**Fig. 3.** HyperShot protocol to request VM snapshots

(assigned by the trusted part of the cloud infrastructure at the VM creation time). In particular, the challenger first creates a non-predictable random nonce $\mathtt{N}$, concatenates it with the $\mathtt{VM_{guid}}$, and sends it to the HyperShot service (1). The nonce guarantees the freshness of the request and *thwarts replay attacks*, whereas the VM identifier *defeats masquerading attacks*.

On receiving a nonce and $\mathtt{VM_{guid}}$, the service front-end finds the physical host on which the $\mathtt{VM_{guid}}$ is running and forwards the request to the HyperShot proxy running on that host. The proxy further forwards the request to the hypervisor and starts the CoW setup process using the hypercall (2). Masquerading attempts in which a malicious root VM modifies the $\mathtt{VM_{guid}}$ that is being passed to the hypervisor can easily be detected by the customer as the $\mathtt{VM_{guid}}$ value will be used to create the final quote by the TPM. Defenses against other forms of masquerading attacks such as “evil clone” are described in Section 5.1.

After initiating the CoW, the forwarder also requests the hypervisor to load the protected key blob for AIK into the TPM, which is used later by the TPM to sign the snapshot (2a-2b). After the CoW setup, HyperShot starts copying the target VM’s memory pages either through CoW faults or by servicing memory copy requests from the forwarder (3). Once the memory copying process is over, the hypervisor generates the composite hash $\mathtt{H_{composite}}$ and associates it with the nonce (4), and stores it for future use.

Next, the forwarder requests a quote over the nonce and the VM identifier from the hypervisor (5). The hypervisor resets PCR 23 (6), and then *extends* it with $\mathtt{H_{composite}}$ corresponding to the nonce and the VM identifier (7):

$$\mathtt{PCR}_{23} = \mathtt{Extend}(0||\mathtt{H_{composite}})$$

The hypervisor then generates the following *TPM_Quote* request (8):

$$\mathtt{TPM\_Quote_{AIK}}(\mathtt{N}||\mathtt{VM_{guid}})[\mathtt{PCRs}]$$

where $\mathtt{AIK}$ is the handle for an $\mathtt{AIK_{priv}}$ already loaded in the TPM, and $\mathtt{PCRs}$ is the set of $\mathtt{PCRs} = \{17, 18, 22, 23\}$ to be included in the quote. As described earlier, PCRs 17, 18, and 22 report the integrity of the platform TCB, while PCR 23 reports the integrity measure for the snapshot.

The generated quote is sent back to the forwarder (9, 9a), which sends it to the HyperShot service front-end, which in turn sends it to the challenger along with the snapshot file and AIK certificate (9b). The snapshot verification process performed at the verifier/challenger site is explained in the next section.

### 4.2 Snapshot Verification Protocol

The verification of the snapshot is straightforward, and it is performed completely in software. In the first phase, the verifier ensures that the quote obtained from the forwarder is valid. This requires completing the following steps:

- The verifier checks that $\mathtt{AIK}_{\mathtt{pub}}$ is indeed a valid AIK based on a certificate associated with the AIK obtained. Next, it verifies the signature from the quote process.
- It determines if the values of PCRs 17, 18, and 22 included in the quote correspond to a known good hypervisor. The known good values of a hypervisor are known to the verifier a priori. This informs the challenger that the hypervisor's integrity was maintained during the snapshot.
- It extracts $\mathtt{H}_{\mathtt{sent}}$ as the value of PCR 23 included in the quote.

In the next phase, the verifier computes the composite hash, $\mathtt{H}_{\mathtt{local}}$, over the memory contents contained in the snapshot file by using the same algorithm as used by HyperShot (described in Section 3.5). Next, the verifier performs the extend operation (in software):

$$\mathtt{H}_{\mathtt{final}} = \mathtt{Extend}(0 || \mathtt{H}_{\mathtt{local}})$$

If $\mathtt{H}_{\mathtt{final}} = \mathtt{H}_{\mathtt{sent}}$, then the snapshot received by the verifier is trusted. Otherwise, the verifier discards the snapshot and takes remedial action, such as informing the provider or moving its work to an alternate provider.

## 5 Evaluation

HyperShot's functionality extends the platform hypervisor with $\sim 4K$ lines of C code, a modest increase compared to the original size of the hypervisor. This design results in a large TCB reduction from the point of view of the snapshot service as HyperShot does not rely on the root VM and its administrators [37]. Next, we provide a detailed security and performance evaluation of HyperShot and its deployment strategy.

### 5.1 Security Analysis and Evaluation

We analyze the security of HyperShot against threats described in Section 2.1, such as tampering, reordering, replaying, and masquerading. We first considered the tampering attack and used the same attack as described in Section 2.2 to compromise the integrity of the snapshot file. After the completion of the snapshot process but before the snapshot is sent to the client, we used Volatility [42] to alter the generated snapshot file stored in the root VM. We deleted a process from the list of running processes. The modified snapshot file and the TPM quote were sent to the verifier, which followed the protocol described in Section 4.2. The verification failed since the hash calculated by the verifier did not match the hash included in the trusted quote. Note that this attack was successful on a system with stock Hyper-V snapshotting in the root VM.

Next, we analyzed an attacker's ability to manipulate the snapshot via manipulation of the proxy. An attacker may launch reordering attacks either by reshuffling the content of the snapshot file or by compromising the code and data of the HyperShot proxy. HyperShot mitigates page reordering attacks by following a simple ordering when generating the composite hash. This design forces the proxy to copy memory pages in the snapshot file in the same order. If this ordering is changed by malware instances or malicious administrators, then the generated composite hash in the hypervisor will not match with the hash calculated at the verifier. HyperShot also prevents other attacks originating from the proxy software inside the root VM. An untrusted root VM may force the proxy to skip the snapshot of some memory pages that they want to hide. To defeat these attacks, the hypervisor ensures that the set of pages protected at the beginning of the CoW setup is the same as the set of pages hashed, and if not, it does not generate the TPM quote over the composite hash.

HyperShot thwarts replaying and masquerading attacks by using a nonce and VM identifiers in the request from the clients. The nonce allows HyperShot to distinguish between a new and the old request, while a VM identifier allows it to identify the target VM. Since the TPM quotes the generated snapshot using the nonce and the VM identifier that it receives in the request, the verifier can check whether the snapshot corresponds to a recent request and for the correct VM.

A different version of the masquerading attack may be launched by the root VM by setting up an "evil" clone of the customer VM with malware in it. The root VM may start this clone along with the customer VM (with a different guid, since the hypervisor enforces the uniqueness of VM guids). Next, the root VM may divert all snapshot requests to the correct VM and actual service requests that operate on customer private data to the evil clone. A variant of the same attack has the root VM shutting down the correct customer VM after a snapshot, starting the evil clone with the customer VM's guid, and forwarding service requests to the clone. These attacks can be mitigated by using a communication protocol enhanced with attestation, such as a quote based on the VM's recent snapshot, in a manner similar to the one proposed by Goldman et al. [13]. They addressed the lack of linkage between the endpoint identity determination and remote platform attestation, which is precisely the root of the evil clone problem. In short, the solution requires generating an SSL certificate for the VM and incorporating the hash of this certificate in the TPM quote, along with the VM's snapshot hash. Both of these values can be captured in vPCRs by our hypervisor.

Our security analysis demonstrates that HyperShot's design effectively mitigates attacks in a cloud environment. Although we have not explored any responses to an untrusted snapshot other than to discard it, it is plausible that in an operational cloud environment, customers would escalate this security threat with the infrastructure provider.

We also evaluated the usefulness of a trusted snapshot by integrating the snapshot generated by HyperShot with two runtime integrity management tools, Volatility [42] and KOP (Kernel Object Pinpointer) [5]. Volatility is an open source forensic tool that analyzes OS memory images. It recognized the snapshot generated by HyperShot without any modifications to its image parsing logic. We extracted information such as a list of running processes, drivers, files, dlls, and sockets from a snapshot of a Windows XP guest VM. With this information, customers could identify potentially malicious software present in their VMs at the time of snapshot, *without the possibility of any tampering from the cloud infrastructure provider.*

KOP uses offline static analysis of a kernel's source code to enable systematic kernel integrity checking by mapping kernel data objects in a memory snapshot. In its current implementation, KOP supports Windows Vista and requires the memory image to be in the *crash dump format*. In our experiment, we ran a kernel malware instance that hides its presence by deleting itself from the list of loaded drivers inside a Vista guest VM. We used an offline utility to convert a HyperShot snapshot into the crash dump format, and we then ran KOP on the dump. KOP successfully found the hidden malicious driver object in the kernel memory. Our integration demonstrates the usefulness of HyperShot as an integral component for managing end-to-end runtime integrity management of VMs in a public cloud environment.

## 5.2 Performance Evaluation

In this section, we present micro- and macro-benchmark results to quantify (1) the performance impact of HyperShot's trusted snapshot generation on target VMs, (2) the demand put on system resources, specifically memory, and (3) the performance of end-to-end snapshot generation and verification. All experiments are performed on an Intel Quad Core 2.53 GHz machine with 4 GB of memory and Extended Page Table (EPT) support [16]. EPT allows HyperShot to modify the GPA-to-SPA map using the second level page table kept in the hypervisor, while the OS inside the VM manages the traditional first level page tables to map virtual address to GPA. For simplicity, we ran the challenger application and the forwarder inside the root VM, which runs 64-bit Windows Server 2008 R2, and obviated the service front-end. The guest VM was allocated 1GB RAM, 1 CPU, and ran 32-bit Windows XP with service pack 3. For macro-benchmark experiments, we used the Passmark Performance Test benchmark suite [29] inside the guest VM.

HyperShot uses CoW to protect memory pages that generate faults whenever the guest or root writes on these protected pages. On each CoW fault, besides changing the page protection, HyperShot also copies the contents of the page if needed. *The median time to perform the additional copy operation is 6.45$\mu$s.*

Next, we measured HyperShot's overhead on different workloads using the PassMark benchmark suite running inside a guest VM. We used 5 representative benchmarks to measure HyperShot's impact on CPU, memory read and write, and sequential disk read and write performance respectively. Each benchmark was executed for 30 times in a loop, and we snapshotted the VM during this loop's execution. Each iteration of the loop ran for a short period of time, so that multiple iterations (but not all) were impacted by the snapshot process. The overall loop's execution encompassed the whole snapshot process. Mean results in Figure 4 indicate that HyperShot imposes no

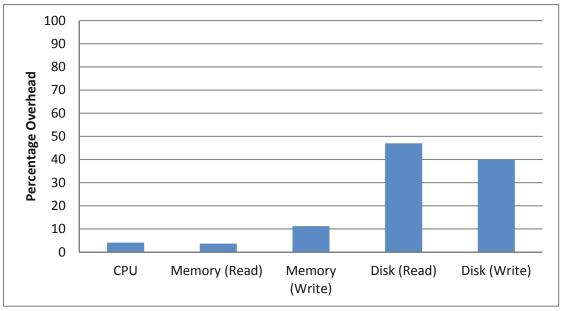

**Fig. 4.** Percentage overhead incurred by HyperShot on various workloads. Smaller measurements are better

discernible overhead on CPU and memory read benchmarks, and a modest $\sim 12\%$ overhead on memory writes. This is expected, since CPU and memory read benchmarks do not generate many CoW faults, while the memory write benchmark generates many CoW faults when it writes data to the memory pages. Higher overheads for disk benchmarks are due to a pro-active, per-page hypercall made by the root VM to copy a page before any DMA operation in anticipation of a CoW fault. This hypercall is made even if the page may already have been copied due to an earlier CoW fault or even if the page will not generate a CoW fault due to a DMA write request (which reads the memory page and writes to the device). This hypercall is serialized with the HyperShot proxy, thereby effectively reducing the parallelism in the system.

We measured HyperShot's CoW memory storage requirements for different workloads during the snapshot of a guest VM. These workloads have different runtime memory requirements and access patterns, and hence provide a good indicator of typical memory usage by HyperShot. We calculate percentage memory consumption as

$$\frac{\text{number of pages copied due to a CoW fault} \times 100\%}{\text{total number of memory pages protected}}$$

The denominator represents an upper bound on the number of pages that will need to be copied in the worst case. The total number of protected pages for a guest VM with 1 GB memory is 262,107.

The results shown in Table 1 are median values for 5 runs of each workload, with the overall as a median of all 30 runs. These results indicate that the memory requirement for HyperShot is $<1.5\%$ for all the workloads considered here, with memory write and disk read benchmarks on the higher side since they generate more CoW faults. This indicates that an alternative, more efficient, strategy for HyperShot would be to deposit only a fraction of the total target VM memory size from the root VM to successfully finish the snapshot, rather then its current strategy of depositing the amount equal to target VM's memory size. If more memory is needed during a guest VM's snapshot, the hypervisor can pause the guest VM (so it does not generate any further faults) and request more memory to be deposited from the root VM. Note that it may not work for all cases, e.g. if the fault is generated due to an access from the root VM, or if the root

**Table 1.** Memory overhead due to CoW fault handling for different workloads in a guest VM

| *Operations* | *Number of CoW Fault* | *% Memory* Consumption |
|---|---|---|
| Idle VM | 1622 | 0.62 |
| CPU workload | 3588 | 1.37 |
| Memory read | 2793 | 1.07 |
| Memory write | 3688 | 1.41 |
| Disk write | 2955 | 1.13 |
| Disk read | 3032 | 1.16 |
| **Overall** | 3407 | **1.30** |

**Table 2.** Time to take SHA-1 hash of a single page of size 4KB and the composite hash on the VM of size 1024 MB

| *Operations* | *Time* |
|---|---|
| Single memory page hash | 149.0 $\mu$s |
| Composite hash | 186.5 ms |

VM itself is being snapshotted. In this case, HyperShot will have to abort and restart the snapshot process.

In our next set of experiments, we quantify the performance of the snapshot generation and verification. First, we micro-benchmarked the time to perform hashing operations. As described earlier, HyperShot performs two different hashes: (a) hashing individual memory pages at the time of memory reads by the HyperShot proxy, and (b) a composite hash at the end of the snapshot process. The results shown in Table 2 indicate that hashing operations are fast, and their impact on overall snapshot generation is small.

Next, we measured the time to finish various stages of the snapshot process from the HyperShot proxy for a guest VM and for an idle root VM. The guest VM was either idle or was executing one of the 5 benchmarks from the PassMark suite described earlier. We measured CoW setup time (initialization time), VM memory copy time, cleanup time, and TPM quote time. Due to brevity, we have skipped the details for each workload type. Not surprisingly, we found the overall time to be dependent more on the number of memory pages being snapshotted and less on the workload itself. *For guest VMs of memory size 1 GB, the overall time to finish the snapshot was 39s, of which 37s were spent in making the memory copy from the hypervisor one page at a time and writing it to the snapshot file, 1s in obtaining the TPM quote, and 1s in initialization and cleanup.* Overall time for the root VM's snapshot was 391s, of which 218s are spent in memory copy, 3s in initialization, 169s in cleanup, and 1s in TPM quote. The increased cost for the root VM is due to both large memory size and a larger number of CoW faults generated by the root VM during its normal execution. Further performance improvements can be made by sharing memory page information between the HyperShot proxy and the hypervisor so that it can avoid hypercalls for pages that are deposited for CoW processing and are not part of snapshot, or have been copied already in DMA setup path.

*For the challenger, generating the hash for a snapshot of 1GB memory took 22s, while the TPM quote validation in software and matching the hash to the one reported in the quote took negligible time.* We assume that the snapshot file and the TPM quote would be transferred out-of-band to the challenger, and the cost of this step would depend on the network bandwidth between challenger and the cloud. As mentioned in Section 4.1, the challenger could be the end-user of the cloud or any third party who has been appointed on the behalf of the customer to verify the integrity of the snapshot.

Although further performance optimizations are possible, results from the prototype implementation show that HyperShot's design is viable, and it imposes moderate overhead on the snapshotted VMs and on the overall platform.

### 5.3 Deployment Strategy

We envision that cloud providers would add the interface for trusted snapshot requests to their web-based VM management front-end. This front-end communicates with the management plane of the cloud infrastructure that controls hundreds of thousands of machines, each running multiple VMs hosting customers code, data, and services. We assume that each physical machine has a TPM chip, which is available today on most commodity hardware platforms. Cloud providers set up each physical machine as it is provisioned to the data center with software and credentials in order to participate as part of the hosting infrastructure. As part of this provisioning, the system software on the machine would generate the AIK using the TPM and would obtain an AIK certificate from a trusted certification authority (CA). This CA is either part of the trusted infrastructure of the cloud provider or an external party, and the certificate ensures that $AIK_{priv}$ is protected by a physical TPM. This certificate is then shared with the HyperShot service front-end. On a request from a customer to generate a trusted snapshot, the trusted infrastructure maps the globally unique identifier (guid) of the VM mentioned in the request to a physical host running the customer's VM. After generating the snapshot, the final quote, the AIK certificate associated with the physical machine, and the snapshot are returned to the customer. The customer can use the certificate to validate the AIK which is then used to verify the integrity of the snapshot. Since the contents of a snapshot are integrity protected, the cloud provider is free to store the snapshot as it deems fit—on a local disk, a network share, or using a cloud storage service such as S3 or EBS.

### 5.4 Discussion

HyperShot is vulnerable to a scrubbing attack where a malicious administrator or compromised root VM can scrub attack traces from the customer VM before the snapshot process starts. A possible solution to this problem is to keep the snapshot request hidden from the malware until the CoW initialization is over. This requires an out-of-band direct communication channel between the service front-end and the hypervisor on the physical machine. It may be possible to establish such a channel over the secondary management communication infrastructure using special purpose processors, such as Intel AMT. The viability of this type of solution has been demonstrated by recent integrity measurement work [1]. An alternative solution makes the snapshot generation an asynchronous process, with the hypervisor initiating the CoW protection at a random point in time. HyperShot only records a VMs' memory and registers; it does not yet snapshot disk contents. We plan to extend it with an existing virtual disk snapshot solution, such as Parallax [23] or the disk snapshotter available with Hyper-V.

## 6 Related Work

Trust issues in cloud computing are an active area of research investigations. Recent work has showed how to use untrusted cloud infrastructures to store and relay information [10, 19]. Trusted computing and virtualization can further enable more general purpose services in a public cloud environment. Santos et al. [34] presented a TPM-based architecture to protect the confidentiality and integrity of data in the cloud. Krautheim [18] proposed a new management and security model using TPMs called private virtual infrastructure (PVI) for cloud computing. TrustVisor [20] provided safety of computation using a TPM based design. Sailer et al. [33] designed and implemented a trusted computing based integrity measurement architecture for desktops. Schiffman et al. [35] proposed a centralized verification service that produces TPM-based attestation of customers' cloud instances for the integrity and transparency purposes. In a similar manner, HyperShot uses trusted computing technologies such as TPM and secure boot to enable a trusted snapshot capability in a cloud infrastructure, where only minimal trust is placed in infrastructure and the hypervisor.

Past research has proposed various mechanisms to reduce the size of trusted computing bases in virtualized environments. Murray et al. [26] proposed a disaggregation based approach for Xen that split dom0 into multiple small privileged VMs, with emphasis on securing interfaces among them. Nova [37] proposed a micro-kernel inspired design for a secure virtualization architecture. It minimized the amount of code in the privileged hypervisor and moved more functionality into service VMs, thereby reducing the core platform TCB. Terra [11] offered security to customer VMs by developing a small trusted virtual machine monitor and provided an isolated closed-box environment to execute sensitive applications. All of these approaches can be readily leveraged by HyperShot by moving the snapshot functionality to its own *service VM*. However, in order to provide similar trust guarantees as our hypervisor-based approach, the TCB for this snapshot service VM must be carefully managed to limit it to just the hypervisor.

Many commercial virtualization platforms provide a VM snapshot facility based on a CoW mechanism [9, 24]. However, these solutions depend upon the correct operation of the root VM, and they rely on potentially malicious administrators to assist in the snapshot process [30]. In contrast to these systems, HyperShot does not rely on the privileged VM and administrators and protects the integrity of a VM's snapshot from unwanted modification from these entities. HyperShot even supports snapshotting of the privileged VM itself using the same mechanisms, a feature missing from current virtualization solutions. These properties makes HyperShot more suitable for a virtualization based public cloud infrastructure.

## 7 Conclusions

We investigated trust issues between cloud customers and providers. To allow customers to establish trust in the public cloud infrastructures, we designed and developed *HyperShot*, a system that securely snapshots a VM even in the presence of a malicious root VM or malicious administrators. HyperShot employs TPM-based *attestation* and

TXT-based *trusted launch of the hypervisor* to protect the integrity of the snapshot via a signature and trusted reporting of the platform state. HyperShot extends the same snapshot functionality and trust guarantees to the root VM itself. This enables cloud infrastructure providers and customers alike to manage the security and integrity of their VMs based on trusted snapshots. We integrated HyperShot with forensic and runtime integrity measurement utilities. Our performance evaluations showed that HyperShot incurs moderate overhead on the VMs' performance.

## References

1. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In: ACM CCS, Chicago (October 2010)
2. Balding, C.: What everyone ought to know about cloud security, `http://www.slideshare.net/craigbalding/what-everyone-ought-to-know-about-cloud-security` (last accessed April 08, 2012)
3. Baliga, A., Ganapathy, V., Iftode, L.: Automatic inference and enforcement of kernel data structures invariants. In: Proc. of ACSAC, Anaheim, CA (December 2008)
4. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: ACM SOSP, NY (October 2003)
5. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping Kernel Objects to Enable Systematic Integrity Checking. In: ACM CCS, Chicago, IL (November 2009)
6. Christodorescu, M., Sailer, R., Schales, D., Sgandurra, D., Zamboni, D.: Cloud Security Is Not (Just) Virtualization Security. In: Proc. of CCSW, Chicago, IL (November 2009)
7. Cihula, J.: Trusted Boot: Verifying the Xen Launch, `http://xen.org/files/xensummit_fall07/23_JosephCihula.pdf` (last accessed April 08, 2012)
8. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live Migration of Virtual Machines. In: Proc. of USENIX NSDI, Boston, MA (May 2005)
9. Colp, P., Matthews, C., Aiello, B., Warfield, A.: VM Snapshots, `http://www.xen.org/files/xensummit_oracle09/VMSnapshots.pdf` (last accessed April 08, 2012)
10. Feldman, A.J., Zeller, W.P., Freedman, M.J., Felten, E.W.: SPORC: Group Collaboration using Untrusted Cloud Resources. In: Proc. of OSDI, Vancouver, Canada (October 2010)
11. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A virtual machine-based platform for trusted computing. In: Proc. of ACM SOSP, NY (October 2003)
12. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proc. of NDSS, San Diego, CA (February 2003)
13. Goldman, K.A., Perez, R., Sailer, R.: Linking remote attestation to secure tunnel endpoints. In: ACM STC, Alexandria, VA (October 2006)
14. Haeberlen, A.: A case for the accountable cloud. In: Proc. of LADIS, Big Sky, MT (October 2009)
15. Intel. Intel Trusted Execution Technology, `http://www.intel.com/technology/security/` (last accessed April 08, 2012)

16. Intel. Intel Virtualization Technology: Hardware support for efficient processor virtualization, http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art01.pdf (last accessed April 08, 2012)
17. King, S.T., Chen, P.M., Wang, Y.-M., Verbowski, C., Wang, H.J., Lorch, J.R.: SubVirt: Implementing Malware with Virtual Machines. In: IEEE Symposium on Security & Privacy, Oakland, CA (May 2006)
18. Krautheim, F.J.: Private Virtual Infrastructure for Cloud Computing. In: Proc. of HotCloud, San Diego, CA (June 2009)
19. Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., Walfish, M.: Depot: Cloud Storage with Minimal Trust. In: Proc. of OSDI, Vancouver, Canada (October 2010)
20. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: TrustVisor: Efficient TCB reduction and attestation. In: IEEE Symposium on Security & Privacy, CA (May 2010)
21. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An Execution Infrastructure for TCB Minimization. In: Proc. of ACM EuroSys, Glasgow, UK (March 2008)
22. Merkle, R.C.: A Digital Signature Based on a Conventional Encryption Function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988)
23. Meyer, D.T., Aggarwal, G., Cully, B., Lefebvre, G., Feeley, M.J., Hutchinson, N.C., Warfield, A.: Parallax: Virtual Disks for Virtual Machines. In: Proc. of ACM Eurosys, Scotland (March 2008)
24. Microsoft. Hyper-V Architecture, http://msdn.microsoft.com/en-us/library/cc768520BTS.10.aspx (last accessed April 08, 2012)
25. Molnar, D., Schechter, S.: Self Hosting vs. Cloud Hosting: Accounting for the security impact of hosting in the cloud. In: Proc. of WEIS, Boston, MA (June 2010)
26. Murray, D.G., Milos, G., Hand, S.: Improving Xen security through disaggregation. In: Proc. of ACM VEE, Seattle, WA (March 2008)
27. Open TC. OpenTC PKI: AIK Certificate Creation Cycle, http://opentc.iaik.tugraz.at/index.php?item=pca/pca_aik_create (last accessed April 08, 2012)
28. Parno, B., McCune, J., Perrig, A.: Bootstrapping Trust in Commodity Computers. In: Proc. of IEEE Symposium on Security & Privacy, Oakland, CA (May 2010)
29. Passmark Software. PassMark Performance Test, http://www.passmark.com/products/pt.htm (last accessed April 08, 2012)
30. Potter, S., Bellovin, S.M., Nieh, J.: Two-person control administration: Preventing administation faults through duplication. In: Proc. of LISA, Baltimore, MD (November 2009)
31. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In: ACM CCS, Chicago (November 2009)
32. Rutkowska, J.: Subverting Vista kernel for fun and profit. In: Black Hat USA (2006)
33. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: Usenix Security, San Diego, CA (August 2004)
34. Santos, N., Gummadi, K.P., Rodrigues, R.: Towards trusted cloud computing. In: HotCloud, San Diego, CA (June 2009)
35. Schiffman, J., Moyer, T., Vijayakumar, H., Jaeger, T., McDaniel, P.: Seeding clouds with trust anchors. In: Proc. of CCSW, Chicago, IL, (Novemer 2010)
36. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In: Proc. of ACM SOSP, WA (October 2007)
37. Steinberg, U., Kauer, B.: NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In: Proc. of ACM Eurosys, Paris, France (April 2010)

38. Trusted Computing Group. TPM Specification version 1.2, Parts 1, 2, & 3, http://www.trustedcomputing.org (last accessed April 08, 2012)
39. VMware. Debugging Virtual Machines with the Checkpoint to Core Tool, http://www.vmware.com/pdf/snapshot2core_technote.pdf (last accessed April 08, 2012)
40. VMware. The Architecture of VMware ESXi, http://www.vmware.com/files/pdf/vmware_esxi_architecture_wp.pdf (last accessed April 08, 2012)
41. VMware. Virtualization Software, http://www.vmware.com (last accessed April 08, 2012)
42. Volatility. The Volatility framework: Volatile memory artifact extraction utility framework, https://www.volatilesystems.com/default/volatility (last accessed April 08, 2012)