Towards Verifiable Resource Accounting for Outsourced Computation

Chen Chen
CyLab, Carnegie Mellon University
Pittsburgh, PA, USA

Petros Maniatis
Intel Labs, ISTC-SC
Berkeley, CA, USA

Adrian Perrig
CyLab, Carnegie Mellon University
Pittsburgh, PA, USA

Amit Vasudevan
CyLab, Carnegie Mellon University
Pittsburgh, PA, USA

Vyas Sekar
Stony Brook University
Stony Brook, NY, USA

Abstract

Outsourced computation services should ideally only charge customers for the resources used by their applications. Unfortunately, no verifiable basis for service providers and customers to reconcile resource accounting exists today. This leads to undesirable outcomes for both providers and consumers—providers cannot prove to customers that they really devoted the resources charged, and customers cannot verify that their invoice maps to their actual usage. As a result, many practical and theoretical attacks exist, aimed at charging customers for resources that their applications did not consume. Moreover, providers cannot charge consumers precisely, which causes them to bear the cost of unaccounted resources or pass these costs inefficiently to their customers.

We introduce ALIBI, a first step toward a vision for *verifiable resource accounting*. ALIBI places a minimal, trusted reference monitor underneath the service provider's software platform. This monitor observes resource allocation to customers' guest virtual machines and reports those observations to customers, for verifiable reconciliation. In this paper, we show that ALIBI efficiently and verifiably tracks guests' memory use and CPU-cycle consumption.

Categories and Subject Descriptors D.4.6 [*Security and Protection*]: Access controls; K.6.4 [*System Management*]: Management audit; K.6.5 [*Security and Protection*]: Unauthorized access

General Terms Measurement, Reliability, Security, Verification

Keywords Cloud computing, Accounting, Metering, Resource auditing

1. Introduction

The computing-as-a-service model – enterprises and businesses outsourcing their applications and services to cloud-based deployments – is here to stay. A key driver behind the adoption of cloud services is the promise of reduced operating and capital expenses, and the ability to achieve elastic scaling without having to maintain a dedicated (and overprovisioned) compute infrastructure. Surveys indicate that 61% of IT executives and CIOs rated the "pay only for what you use" as a very important perceived benefit of the cloud model and more than 80% of respondents rated competitive pricing and performance assurances/Service-Level Agreements (SLAs) as important benefits [3].

Despite this confirmation that resource usage and billing are top concerns for IT managers, the verifiability of usage claims or services provided has so far received limited attention from industry and academia [34, 39]. Anecdotal evidence suggests that customers perceive a disconnect between their workloads and charges [1, 4, 12, 29]. At the same time, providers suffer too as they are unable to accurately justify resource costs. For example, providers today do not account for memory bandwidth, internal network resources, power/cooling costs, or I/O stress [22, 30, 46]. This accounting inaccuracy and uncertainty creates economic inefficiency, as providers lose revenue from undercharging or customers lose confidence from overcharging. While trust in cloud providers may be a viable model for some, others may prefer "trust but verify" given providers' incentive to overcharge. Such guaranteed resource accounting is especially important to thwart demonstrated attacks on cloud accounting [27, 42, 50].

Our overarching vision is to develop a basis for *verifiable resource accounting* to assure customers of the absence of billing inflation, thereby forestalling billing disputes. Furthermore, the enhanced transparency of precise resource accounting helps cloud users optimize their utilization.

Unfortunately, existing trustworthy computing mechanisms provide limited forms of assurance such as launch integrity [40] or input-output equivalence [18], but do not address resource accounting guarantees. An alternative is to develop "clean-slate" solutions such as a new resource-accounting OS or hypervisor [25]; however, these are not viable given the existing legacy of deployed cloud infrastructure.

The challenge here is to achieve verifiable resource accounting with *low overhead* and *minimal changes* to existing deployment models. To this end, we propose an architecture that leverages recent advances in *nested virtualization* [9, 48]. Specifically, we envision a thin lightweight hypervisor atop which today's legacy hypervisors and guest operating systems can run with minor or no modification. Thus, this approach lends itself to an immediately

VEE'13, March 16–17, 2013, Houston, Texas, USA.

Reprinted from VEE'13,, [Unknown Proceedings], March 16–17, 2013, Houston, Texas, USA., pp. 1–12.

deployable alternative for current provider and customer side infrastructures.

The properties of verifiable resource accounting, however, do not directly map to the applications targeted by nested virtualization (e.g., defending against hypervisor-level rootkits or addressing compatibility issues with public clouds). Thus, we need to identify and extend the appropriate resource allocation "chokepoints" to provide the necessary hooks, while guaranteeing that customer jobs run untampered.

As a proof-of-concept implementation, we demonstrate verifiable resource accounting by extending the Turtles nested virtualization framework [9], in which we build a minimal trusted *Observer*, observing, accounting for, and reporting resource use. As a starting point, we show this for the two most commonly accounted resources, CPU and memory, which are directly observable by lower virtualization layers, thanks to existing virtualization support in hardware.

Our prototype, ALIBI, is limited and is intended as a proof of concept of verifiable accounting. It demonstrates that: (i) verifiable accounting is possible and efficient in the existing cloud-computing usage model; (ii) nested virtualization is an effective mechanism to provide trustworthy resource accounting; and (iii) a number of documented accounting attacks can be thus thwarted. Our evaluation of the salient points of our system shows that resource accounting and verifiability add little overhead to that of nested virtualization, which is already efficient for CPU-bound workloads. While there is non-trivial overhead for I/O bound workloads, recent and future advances in virtualizing or simplifying interrupts [17], as well as hardware support for nested virtualization [37] make the approach promising.

While ALIBI already represents a significant advance over the status quo in resolving the uncertainty in resource accounting, we acknowledge that this is only a first step. Beyond the aforementioned performance limitations of nested virtualization for I/O-intensive workloads, we need to address several other issues to fully realize our vision for verifiable accounting. As future work, we plan to extend our framework to handle other charged resources, such as I/O requests or provider-specific API requests (e.g., Amazon S3), which are most often not directly observable by the low layers of virtualization. While we expect non-trivial challenges in addressing these issues, the initial success demonstrated here, the experiences we gained in the process, and emerging processor roadmaps give us reasons to be optimistic in our quest.

2. Motivation

In this section, we survey the landscape of outsourced computation, identify shortcomings in how resources are invoiced, and derive the desirable properties for addressing those shortcomings.

2.1 The Lifecycle of Outsourced Computation

The typical outsourced-computation pattern we study in this work is *Infrastructure as a Service* (IaaS), exemplified by Amazon's Elastic Compute Cloud (EC2)[1], Rackspace[2], and Azure[3] among others. IaaS offers customers a virtual-hardware infrastructure to run their applications.

A new customer starts by creating an account on the platform, and exchanging private/public key-pairs, to be able to authenticate and encrypt future communication channels. After account establishment, a customer can upload a virtual-machine image to platform-local storage, which contains a virtual boot disk with an OS, needed applications, and data. The platform operator may require mild customization of that image to improve performance or compatibility, e.g., installing customized device drivers or BIOS. The customer then launches an *instance*, by booting that customized image in a platform guest VM, and either directly logs into that instance to manage it, or lets it serve requests from remote clients (e.g., HTTP requests). While her instance is running, the customer may use additional hosting features, such as local storage (e.g., Amazon's Elastic Block Store (EBS)[4]). Later on, the customer terminates that instance.

The platform provider charges the customer either for provisioned services or according to usage. For example, EC2 charges a customer for the total time her instance is in a running state (length of time between launch and termination, even if the virtual CPU is idle in between). Additionally, EC2 charges the customer per distinct I/O request sent by her instance to a mounted EBS volume[5]. The former is an instance of a provisioned service, charged whether it is used or not, while the latter is an instance of a pay-per-use service. Although platform operators provide some SLAs (e.g., Amazon offers a minimum-availability guarantee[6], and a credit process when that guarantee is violated during a pay cycle), most provisioned services (e.g., a provisioned-IOPS EBS volume, which has a provisioned bandwidth of up to 1000 I/O operations per second) are not accompanied by precise SLAs. Except for small differences, other providers, such as Microsoft's Windows Azure service, operate in a similar fashion for their IaaS products.

To summarize, the lifecycle of a customer's VM on a provider's platform has the following steps: (i) Image installation; (ii) Image customization; (iii) Instance launch of an installed image; (iv) Execution accounting of resource use by the instance; (v) Instance termination; and (vi) Customer invoicing based on instance-usage accounting.

2.2 Challenges with Unverified Resource Use

We now identify how lack of verifiability can cause accounting inaccuracy and deception in the context of the outsourced-computation lifecycle.

Image Installation The transfer of a new VM image from the customer to the platform incurs network costs, and the storage of an installed image incurs storage costs. If the installation channel lacks integrity guarantees, external attackers may cause extraneous storage and network charges. In fact, the management interfaces of both EC2 and Eucalyptus, an open-source cloud-management platform, were found vulnerable to such abuse, making this a realistic threat. Somorovsky et al. [42] used variants of XML signature-wrapping attacks [28] to hijack the command stream between a legitimate customer and the provider. In this fashion, an attacker may replace the image installed by a customer and cause subsequent launches to bring up the wrong image.

In a similar fashion, the provider is currently unconstrained from performing image installation; e.g., by discarding the image supplied by the customer and replacing it with another. This is a special case of outsourced-storage integrity and retrievability [41].

Image Customization Before execution, a customer's image may be modified for the hosting platform. For example, the provider may install its proprietary drivers or BIOS into the image. This may constitute a legitimate reason why the image that runs in the cloud is different from the customer-supplied image. Furthermore, the provider may wish to conceal proprietary information about its platform and its customizations.

[1] `aws.amazon.com/ec2/`

[2] `www.rackspace.com`

[3] `www.windowsazure.com`

[4] `aws.amazon.com/ebs/`

[5] `aws.amazon.com/ec2/pricing/`

[6] `aws.amazon.com/ec2-sla/`

Instance Launch A launch event (i.e., when an image is launched within a VM instance) is significant for accounting purposes – this is the time when actual charges start accruing for on-demand pricing schemes. Unfortunately, nothing stops a greedy provider from spuriously starting an instance and there is no defense against external attackers who abuse the control interfaces [42] to start an instance on behalf of an unsuspecting customer.

Execution Accounting There is little a customer can do to ensure that, after launch, her instance continues to run the intended image; e.g., the platform or an external attacker can suspend the instance, replace its image with another, and resume it. Practical attacks have been demonstrated against the prevalent (sampling-based) scheduling and accounting where malicious customers can run their own tasks but cause charges to be attributed to other customers. One such attack, described by Zhou et al. [50], allows instances that share a physical CPU to suspend themselves right before a scheduler tick is issued. As a result, the victim customer's instance that is subsequently scheduled gets charged for being active during the scheduler tick.

On the other hand, platform providers, even when promising dedicated resources, can inflate charges. For example, larger EC2 instances (e.g., a "Medium" instance) are assigned – and charged – dedicated CPUs and memory while the instance is running. But a customer may wonder if the CPU she is paying for is really dedicated; can a provider *overbook* (or, more bluntly put, *double-charge*) by "dedicating" the same physical CPU to multiple instances?

Liu and Ding have identified ways in which a platform provider can subvert the integrity of resource metering [27]. Even assuming limited attack capabilities – in their case, an attacker who can only change privileged software but not system software or the customer's image – a malicious provider can inflate resource use by arbitrarily prolonging responses to the customer instance's system requests. Such requests include the setup period between instance launch and control transfer to the customer's image; the handling of system calls, hypercalls, exceptions, and I/O requests; the issuance of extraneous interrupts; and the implementation of platform features in local or remote libraries.

Instance Termination Termination is the end point of the CPU-charging period for instances and, consequently, it is another critical event for proper accounting. Premature termination of an instance (e.g., against the customer's intentions) may indicate the replacement of the image in a running instance with another arbitrary one. Also, delayed termination past the point dictated by the customer or her management scripts may be an avenue for deceptively inflating usage charges.

Invoicing The invoice generated by the provider and submitted to the customer for payment is intended as a summarized record of the customer's use of the provider's resources. The challenge with verifiable accounting is to ensure that this record is consistent with the actual usage incurred by the customer's VMs. For example, an external attacker, especially one with unchecked access to the management interface, may pass her own use of the platform as incurred by a different customer. Conversely, the platform operator may generate inflated invoices, since customers cannot witness the usage of their own instances, to associate the invoice with the actual expenditure.

3. Desired Properties

The implication of the above weaknesses is that the customer who receives an invoice at the end of a billing cycle cannot distinguish between charges for her legitimate VM image, or some attacker-installed VM image running on her behalf, or charges arbitrarily and undeservedly assessed by a deceitful provider. Building on the attack scenarios described above, we identify three properties: Image Integrity (*what* is executing), Execution Integrity (*how* it is executing), and Accounting Integrity (*how much* provider charges customer). To achieve *verifiability*, a customer needs assurance that the provider cannot violate the integrity properties undetected and, conversely, a correct provider needs assurance to avoid slander for purported integrity violations.

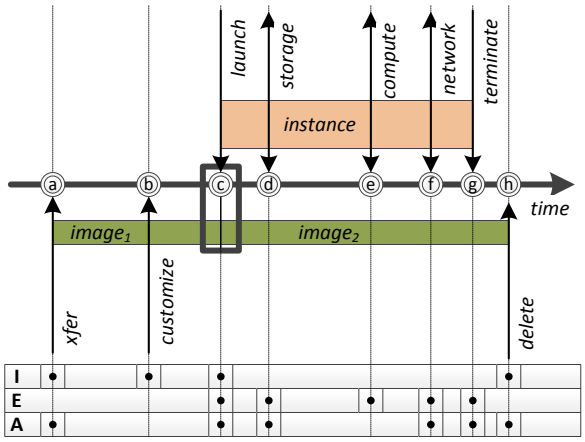

Figure 1. The System Model: There are three types of integrity properties Image (I), Execution (E), and Accounting (A). The figure shows a timeline during the lifecycle of an outsourced computation task and how different events relate to the integrity properties we require for verifiable accounting.

To formulate these properties, we consider the system model illustrated in Figure 1. The customer-provider interface includes operations to transfer new images (*xfer*), to *customize* images before launch, and to *delete* images from storage, to stop incurring storage costs. Instances can be *launch*ed using a previously-installed image, and *terminate*d later on. While an instance is running, it undergoes state changes, including requests for *storage*, *network*, and *compute*. Some of these operations are relevant to images (I), some to execution (E), and some are chargeable events relevant to accounting (A), as shown at the bottom of the figure.

Image Integrity Informally, the OS, programs, and data making up the customer's *image* must have the contents intended by the customer at the time of each instance launch. In other words, the sequence of management operations – image installation, image customization, and instance launch given an image – have the same effect on instance launches (i.e., cause the same image to boot upon instance launch) as they would have if the customer were executing these operations on a trusted exclusive platform.

Note that this property can be maintained while the provider modifies customer images without explicit customer authorization (e.g., by moving them from block device to block device, compressing them, deduplicating them, copying them, etc.). The requirement is that upon a customer-initiated launch, the launched image is as the customer intended via her explicit operations.

Execution Integrity Similarly, changes to the *state* of an image while it is executing in an instance are "correct" if the sequence of actions (instruction execution, requests received externally, non-deterministic interrupts) taken by an image instance between launch and termination have the same effects on the instance state (its local storage while it is running), and external interfaces (e.g., responses sent to remote requests) as it would have, if that

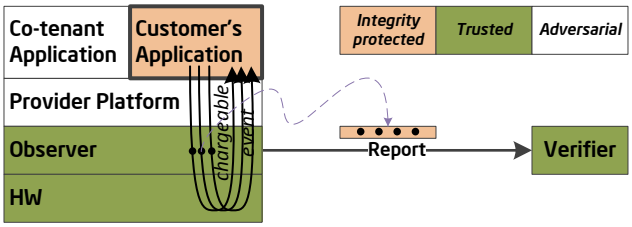

Figure 2. The conceptual architecture of ALIBI. We envision a lightweight trusted Observer that runs below the cloud provider's platform software. This trusted layer generates an attested report or witness of the execution of the guest VM to the customer.

same image were executing under the same sequence of actions on a trusted, correct, exclusive platform.

Since all external devices are under the control of the platform, execution integrity cannot prevent network packets or disk blocks from being malicious, or triggering non-control data vulnerabilities [10]. Integrity here assumes a correct CPU and memory system. This property does not restrict platform operations from suspending an instance, migrating it, or otherwise manipulating it, as long as those manipulations do not alter the behavior of the instance.

Accounting Integrity This property ensures that the customer is only charged for *chargeable* events, such as CPU-cycle utilization, while an instance is running. In other words, the provider cannot charge the customer for spurious events (e.g., for having used a CPU cycle while another instance was using it). Similarly, the property ensures that the customer cannot incur unaccounted chargeable events.

A charging model (i.e., a specification of what events should be charged how much), maps a sequence of image and execution actions to an invoice. Accounting integrity then ensures that the provider invoices the customer as if the customer had run her sequence of actions on a trusted, exclusive platform, and applied the charging model on the resulting action sequence[7].

Verifiable resource accounting requires us to satisfy *all three properties*. With accounting integrity alone, the customer may know that the right events were measured in the invoice (i.e., she was not charged for fictitious cycles), but she cannot know if those events corresponded to her jobs. For that, it is essential to ensure the *correct* execution of the *right* image (execution and image integrity, respectively). Similarly, image integrity alone is meaningless; the provider may charge for arbitrary, spurious events that have nothing to do with the customer's image and precluding that scenario requires accounting integrity. Image integrity, even with accounting integrity, is insufficient, since the provider may inject arbitrary code charges for correct events issued by an instance launched with the correct image, albeit for an incorrect execution.

4. ALIBI Design

The conceptual architecture of our system, ALIBI, is shown in Figure 2. At a high level, ALIBI uses nested virtualization to place a trusted Observer at the highest privilege level underneath the provider's platform software and all customer instances. The Observer collects all chargeable events incurred by a customer instance, and offers them to the customer, as a *trustworthy witness* of the provider's invoice at the end of a billing cycle. At the same time, the Observer protects the execution of the customer instance against tampering by other instances or by the provider itself, while ensuring that the provider does not miss customer actions that it should be charging for.

[7] Charging functions may not be independent from other concurrent users of the platform (e.g., some resources may have congestion pricing, as for example Amazon does with EC2's spot instances). We narrow our scope here to simpler, independent-charge models.

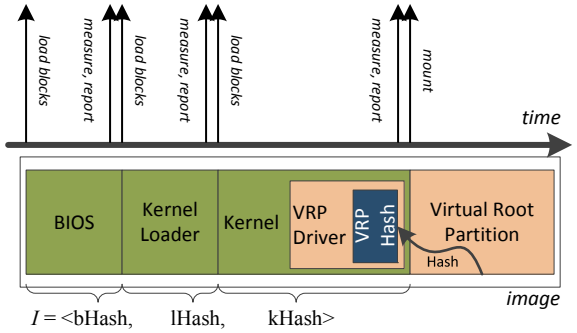

Figure 3. Instance Attestation: Timeline of instance launch showing the different hashes of the BIOS, kernel loader, and kernel being computed in sequence.

We consider two case studies of resource accounting:

CPU Usage The customer agreed to be charged while her application is executing on the provider's CPUs, but not when it is suspended.

Memory Utilization The customer agreed to be charged for the amount of physical memory her applications use, e.g., as the number of pages integrated over allocated time.

In the next sections we explain in order how ALIBI guarantees the three integrity properties from Section 3. Image integrity is protected via *attested instance launch* (Section 4.1). Execution integrity is protected via *guest-platform isolation* (Section 4.2). Accounting integrity is protected via *trustworthy attribution* (Section 4.3). Trust in the operation of the Observer itself is established via *authenticated boot* (Section 4.4). We revisit the lifecycle of an outsourced computation in Section 5, arguing that the weaknesses we identified earlier (Section 2.2) are removed by ALIBI.

Viewed in a general systems context, ALIBI builds on the well-known concepts of reference monitors and virtualization. Our contribution lies in the careful extension of these ideas to meet the particular integrity requirements of verifiable resource accounting.

4.1 Image Integrity via Attested Instance Launch

Image integrity requires that the Observer verify the customer's image when it is first loaded into an instance by the provider platform. If an image were loaded directly and entirely into a sufficient number of memory pages, then the Observer could measure those pages – i.e., hash them in a canonical order with a cryptographic hash function – and compare them to a hash submitted by the customer during image installation.

Unfortunately, VM images are almost never entirely in memory. Although the kernel remains pinned in (guest) memory, user-space processes are placed into guest virtual memory on demand, in the somewhat unpredictable order of process launch, via the init process or the shell. Furthermore, memory-page contents may be swapped out by the instance OS to reuse guest physical memory, or even by the platform provider, to reuse host physical memory, when managing multiple concurrent instances on the same physical hardware (e.g., via ballooning or transcendent memory).

To address this problem, ALIBI uses a hybrid software attestation approach. As in prior systems that bring up an attested kernel (e.g., SecVisor [40]), the customer's BIOS, kernel boot loader,

and kernel are measured and launched one after the other. All remaining data are loaded from the installed image by mounting it in an integrity-protected fashion, either at the file system level, or the storage block device level; protection is done in a traditional chained-hash mechanism (e.g., SFS-RO [16] and dm-verity [2]), and the root hash is hard-coded in the device driver, which is itself statically compiled into the attested kernel. Figure 3 illustrates the image structure.

These properties are guaranteed as follows. The Observer is told explicitly about the $\mathcal{I} \doteq \langle \mathrm{bHash}, \mathrm{lHash}, \mathrm{kHash} \rangle$ triple, containing the cryptographic hash of the BIOS, the kernel loader, and the kernel, respectively, when a new customer image is installed in the platform. Each successive stage of the instance boot process registers itself with the Observer (via a hypercall), reporting what customer image it belongs to (a customer-configured ID), what stage it is (BIOS, loader, kernel), and what guest physical memory pages it occupies; the Observer hashes those memory pages, matches the hash against the corresponding component of $\mathcal{I}$ for that image ID, and records the memory range as part of the instance for the given image.

Once the instance kernel is registered and loaded, it mounts its root partition using the integrity-protected filesystem driver. Recall that the root hash for the file system is embedded in the kernel (as part of the statically compiled device driver), so kHash protects the root partition as well.

At the end of this process, the Observer knows the memory pages occupied by the static and dynamic portions of the customer instance, and that their contents are consistent with the customer's registered image.

4.2 Execution Integrity via Guest-Platform Isolation

ALIBI provides execution integrity by protecting three assets of the running customer instance: its state in memory, its state in storage, and its control flow.

Memory: Given a current allocation of physical memory pages $\mathcal{M}$ to an instance i, the Observer enforces the invariant that memory in $\mathcal{M}$ can only be written while i is executing.

ALIBI enforces this invariant via the Memory Management Unit (MMU), and in particular the Extended Page Tables (EPTs) on Intel processors. An EPT maps guest physical pages to host physical pages, and associates write/execute permissions with each such mapped page, much like traditional page tables. When a guest attempts to access a guest physical page that has not as yet been mapped to a host physical page in the EPT, an EPT violation trap gives control to the hypervisor, which performs the mapping, and resumes the guest. In our case, the Observer write- and execute-protects all pages in $\mathcal{M}$ by modifying the platform software's EPT, while i is not executing. When the platform software attempts to pass control to the customer instance, the instance's EPT will be installed, which automatically unprotects pages in $\mathcal{M}$. When the instance loses control, e.g., because of a hypercall or an interrupt, the Observer automatically re-protects $\mathcal{M}$ by installing the platform software's EPT again.

When an instance is first launched via the mechanism described in Section 4.1, the Observer only associates with the instance the memory pages holding content that has been measured and matched against the image integrity digest $\mathcal{I}$. To capture further modifications of $\mathcal{M}$, the Observer also write-protects the memory-management structures of the platform software. This ensures that the Observer interposes (via EPT violation traps) on all modifications of memory allocations by the platform to its guests. The Observer applies the protection described above to $\mathcal{M}$ as that changes over time, since changes are always essentially effected by the Observer first.

One subtle issue here is that the platform software may have legitimate reasons to modify the contents of a guest's page unbeknownst to that guest, e.g., when migrating the guest to another physical machine, or swapping guest-physical pages out or back in again. While the above protections ensure that the Observer prevents the platform from manipulating guest pages when they are in memory, it does not prevent the pages from being arbitrarily modified when they are swapped out and then swapped back in. This requires an additional, but straightforward, protection. Specifically, when the provider platform needs to unmap a guest physical page (e.g., to swap it to disk), the Observer intercepts this request as above (since all modifications to the guest's EPT by the platform result in a protection trap down to the Observer). At this time, it computes a cryptographic hash of the contents, and records the hash for the guest page address. If the platform later maps another physical page to the same guest page, the Observer once again interposes on this call to check that the contents have not been modified, by checking if the hash matches the recorded value. Manifests of such page hashes can be transmitted to remote Observers during migration. Our prototype does not yet implement this protection.

Note that platform software may write-share memory pages with an instance (and instances may also share pages with each other). We require the guest to explicitly mark some of its pages as authorized for sharing with the platform, and exclude them from the protection described above. For read-shared pages, as might happen, for example, with the Kernel Samepage Merging (KSM) mechanism in Linux-KVM, our protections still apply with appropriate manipulation of the relevant EPTs when the platform attempts to map the same page to multiple guests.

Storage: Instances typically have at their disposal some local storage (EC2 calls it "instance storage") for their lifetime. ALIBI protects that storage by mounting it via an integrity-protected filesystem (a read-write variant of dm-verity [2]), in a manner similar to how the root partition is mounted. Although the mutability of this storage makes integrity protection somewhat more expensive for a naïve implementation, systems such as CloudVisor [48] have demonstrated acceptable performance for even stronger protection of this form (adding confidentiality).

Control Flow: To protect the control flow of instances, ALIBI protects the stacks of a guest (both user-space and kernel-space) as part of protecting the allocated memory pages to an instance. As a result, the call stacks of processes in the instance cannot be directly altered by the platform or other instances.

While an instance is not running, platform software has control of the guest-CPU state, including the stack-pointer and instruction-pointer registers (RSP and RIP), which also affect control flow when the instance resumes, as well as general-purpose registers, which may indirectly affect control flow upon resumption, and model-specific registers, which may affect the general operation of an instance (e.g., disable memory paging). ALIBI uses memory protections on the data structures holding guest state in the platform software, after an instance is launched; when platform software attempts to modify such state, the Observer validates the modification before allowing it to affect guest operation.

In general, ALIBI limits the options of in-flight modifications of guest state available to the platform. In particular, it only allows changes to RSP and RIP that are consistent with handling of guest-mode exceptions (e.g., emulated I/O requests) that, typically, amount to advancing the RIP register to the next instruction following the one that caused an exception. ALIBI also explicitly records general-purpose registers holding return data from a hypercall, and allows the platform software to modify those registers.

Finally, the control flow of an instance may be affected by the initial instruction executed when the instance is launched (in the

BIOS segment of the image). ALIBI only allows a newly launched instance to be started at a given, fixed initial entry point (typically, the entry point into the BIOS). Subsequent stages in the bootstrap process are protected as described above.

4.3 Accounting Integrity via Bracketing

Accounting integrity relies on three fundamental components, all of which must be verifiable to both parties' satisfaction: (a) chargeable-event detection, (b) chargeable-event attribution, and (c) chargeable-event reporting. Event detection (Section 4.3.1) must ensure that only real events are captured (which precludes spurious charges), and no real events are missed (which precludes service theft). Event attribution (Section 4.3.2) must verifiably associate a detected event with a customer to charge. Finally, event reporting (Section 4.3.3) must protect the collected information at rest on the provider's infrastructure, and in transit to customers.

4.3.1 Event Detection

In this work, we focus on chargeable events that are directly observable by the Observer. For example, given the protections required for image and execution integrity (Sections 4.1 and 4.2), the Observer sees every transfer of control (and, therefore, of the CPU) between the platform software and customer instances. Similarly, the Observer sees every memory allocation and deallocation by the platform software to customer instances. We defer to future work those chargeable events that are not necessarily observable by the Observer, such as I/O requests, especially for directly-assigned devices.

Such direct detection is effective for both instantaneous charging events (e.g., requests for growing a guest's memory footprint) and time-based charging events (e.g., duration of CPU possession by a customer instance). For time-based events, the Observer collects instantaneous events denoting the beginning and end of possession of a chargeable device, from which the duration can then be computed easily (e.g., using clock time, a cycle counter, or other monotonically increasing hardware performance counters).

4.3.2 Event Attribution

Verifiable attribution implies that the provider cannot charge customers for chargeable events willy-nilly, but is bound to charge the customer whose image incurred the event.

ALIBI builds its verifiable attribution machinery on CPU ownership. Because of the attested instance launch mechanism (Section 4.1), the Observer can associate definitively a set of memory pages with a given installed image. Consequently, the Observer can attribute ownership of the CPU to a given image when the CPU enters the pages associated to that image. This means that ALIBI can attribute events that acquire or relinquish ownership of other resources to the appropriate customer image that currently holds the CPU.

4.3.3 Event Reporting

Verifiable reporting implies that the provider cannot report incorrect chargeable-event measurements to the customers, but must report accurate values.

The ALIBI Observer collects event measurements (e.g., CPU possession and guest memory footprint) during the entire lifetime of the customer image execution. The Observer then packages these measurements along with the attestation triplet for a customer image (from Section 4.1) in a signed report that also includes the platform software state (see Section 4.4 below). Finally the Observer ships the signed report to the related customer along with an invoice.

4.4 Trust via Authenticated Boot

Fundamental to any security property that can be ascertained external to a platform manifesting the property of interest is a *root of trust*. ALIBI relies on a Trusted Platform Module (TPM) [5] on the provider platform for this purpose.

At a high-level, the TPM can be thought of as possessing a public-private key-pair, with the property that the private key is only handled within a secure environment inside the TPM. The TPM also contains Platform Configuration Registers (PCRs) that can be used to record the state of software executing on the platform. The PCRs are append-only, so previous records cannot be eliminated without a reboot.

Initially ALIBI is started via a dynamic root of trust mechanism on the provider platform. This can be done for example, by using a trusted boot-loader such as `tboot` [7] or `oslo` [24]. The authenticated boot mechanism ensures that integrity measurements are taken of all loaded code modules. These measurements are extended into one or more PCRs, so that a history of all modules loaded is maintained and cannot be rolled back.

With accumulated measurements from authenticated boot, the root of trust for reporting (or commonly called an attestation) becomes useful. ALIBI uses the TPM to generate an attestation, which is essentially a signature computed with the TPM's private key over some of the relevant PCRs. Given the TPM's corresponding public key, an external verifier can check that the signature is valid and conclude that the PCR values in the attestation represent the software state of the platform (i.e., a correctly loaded ALIBI hypervisor). Note that numerous solutions exist to obtain the TPM's authentic public key [31]. One straightforward approach is to obtain a public-key certificate from the provider which binds the public key to the provider identity.

5. Lifecycle of a Verifiably Accounted Job

As discussed previously, the design of ALIBI makes one practical assumption about the nature of IaaS deployments. In order to assure the customer that the Observer itself was running, we assume that a hardware root of trust, i.e., a TPM chip, is present on the platform and provisioned with appropriate cryptographic material by the manufacturer; this assumption is reasonable given the increasing availability of server-grade hardware platforms equipped with trusted-execution features[8] and the emergence of high-assurance cloud-service solutions such as that by Enomaly[9]. We now review the lifecycle of an outsourced job with ALIBI and highlight how ALIBI addresses the accounting vulnerabilities from Section 2.2.

Image Installation When a customer installs a new VM image, she provides a random nonce, along with the integrity triple $\mathcal{I}$, and only presumes the installation successful upon receiving a receipt containing the triple, the nonce, and a signature on the two from the Observer. Even though the customer may not be directly contacting the Observer, but may instead be using the platform API or web interface, a receipt from the Observer indicates the latter has identified a particular VM image as protected. The nonce protects the installation channel from replay attacks, and the signature protects the communication between the customer and the Observer from the intervening platform software.

Image Customization Customization may result in changes to the customer's image, but is transparent to ALIBI. When a customer is

[8] `http://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-server-platforms-matrix.html`

[9] `http://www.enomaly.com/High-Assurance-E.484.0.html`

done modifying an image, she must reinstall it, as described above, possibly uninstalling the original version of the image.

The explicit re-installation of a customized image prevents surreptitious image modifications before launch, which would be otherwise open to the platform and external attackers hijacking the control interface.

Note here that in this work, we assume the "easier" version of customization, where the platform provider may recommend certain stock device drivers (e.g., paravirtualized Xen device drivers) that must be installed and the customer explicitly and manually installs those drivers to its image before launching. As such, we assume that the stock device drivers are as trusted by the customer as the rest of her image software. We leave for future work the "harder" version of customization, where image modifications are not trusted (e.g., may come in binary form from the provider), which may require more complex solutions, perhaps akin to OSck [20], adapted to the outsourced domain.

Instance Launch Launch for a particular installed image works as described in Section 4.1. The attested instance-launch mechanism ensures that instances are launched legitimately only with full visibility to the Observer, only from images that have been explicitly installed by the customer. What is more, this mechanism ensures that the launch-point state of an instance is consistent with the image, and cannot be modified undetected by the platform.

Execution Accounting During instance execution, integrity is guaranteed through the state and control-flow protections described in Section 4.2. Consequently, surreptitious modifications of system libraries or the internal functionality of an instance [27] are not possible.

The Observer accounts for CPU and memory as described in Section 4.3, and has full visibility of related chargeable events. Although the platform can delay the execution of operations in platform software on behalf of the customer (e.g., the handling of hypercalls issued by the instance), this happens outside the CPU-control of the instance, does not constitute a chargeable event, and is therefore immaterial to the customer's invoice. Note, however, that in a model that charges customers for system costs, this might be more complex to handle, as we describe in Section 8.

Similarly, scheduling tricks [50] have no effect, since charging is done via explicit counting of events, rather than the bias-prone sampling. This also means that the platform cannot charge two customers for the "same CPU cycle" since the CPU instruction pointer can only be in one memory location at a time, and the Observer keeps track of the memory footprint of an instance via its EPT.

Instance Termination When an instance terminates, a running period ends and the platform explicitly deregisters an image from the Observer, thereby removing the physical pages it had previously allocated to that image from the Observer's protection. No (execution-related) chargeable events are collected for that image beyond instance termination.

Invoicing When invoicing the customer, the platform also presents a witness report (Section 4.3.3) consisting of Observer-signed event traces supporting that invoice. Those traces are periodically passed to the platform as the Observer collects them, to minimize the storage requirements for the Observer, but the platform must accumulate and supply those traces to the customer along with an invoice.

The witness report is associated with the precise image that was launched and protected during runtime by the Observer. As a result, an invoice for charges substantiated with a witness generated by an image that the customer did not install can easily be detected as fraudulent.

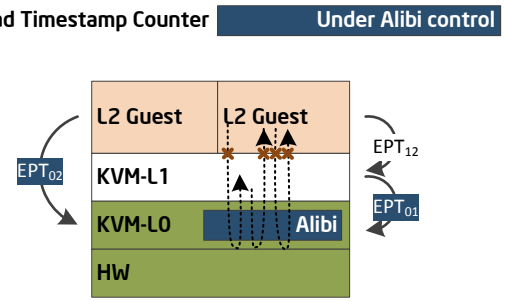

Figure 4. ALIBI Implementation: We currently leverage the nested virtualization support provided via the Turtles project in KVM. ALIBI is a lightweight extension to this nested virtualization codebase. While our current prototype runs KVM as the L_1 hypervisor, this is purely for convenience and does not represent a fundamental constraint.

6. Implementation

In this section, we describe the pieces of the ALIBI prototype we have implemented, and demonstrate the salient aspects of the design from Section 4.

As shown in Figure 4, we have implemented ALIBI on the open-source Linux-KVM hypervisor codebase. Our prototype is based on the Linux-KVM kernel, version 3.5.0, with support for efficient nesting provided by the Turtles developers as separate patches, as yet unincorporated into the mainline kernel. For the purposes of our prototype, we assume that the platform already uses KVM as its virtualization software, and that customer guests run the Linux OS. We implement the ALIBI Observer using another layer of KVM virtualization, below the purported provider's KVM software platform.

We chose KVM because of its advanced and efficient support for nested virtualization [9] on top of modern CPUs' hardware-virtualization features[10]. Although this support is not part of our contribution, we review it in Section 6.1, since it forms the basis for ALIBI's implementation. Then we describe how we implement the particular kind of isolation that is essential for ALIBI's integrity, in Section 6.2. We delve into the implementation details for providing accounting for the two types of resources in Section 6.3. In describing our implementation, we describe specifics pertaining to the Intel platform we use for prototyping; analogous support exists on AMD platforms as well.

6.1 Background: Nested Virtualization with KVM

The basic tools offered by hardware support for virtualization are CPU-state and physical-memory virtualization. Intel-architecture processors virtualize CPU state by providing a data structure in physical memory, called the Virtual Machine Control Structure (VMCS) in Intel's processors, where the host's state is held while a guest is executing, and where the guest's state is held while the host is executing. The VMCS also holds configuration information about what the guest is allowed to do (e.g., which privileged instructions it may invoke without trapping to the host, etc.).

Physical memory is virtualized via an extra layer of page tables, which are called Extended Page Tables (EPT) for Intel's processors; the EPT maps guest physical addresses (GPA) to host physical

[10] On a pragmatic, but slightly non-technical note, we chose KVM because the Turtles code is publicly available. The mechanisms we envision can also be incrementally added to other nested virtualization platforms such as CloudVisor [48]. Unfortunately, the CloudVisor authors could not yet provide us with the source code when we requested it.

addresses (HPA), and can contain read/write/execute protections for mapped pages separate of those in the regular OS-managed page table maintained by the guest. If the CPU attempts to access a GPA in violation of the EPT, the CPU traps from guest to host mode with an `EPT_VIOLATION` exception.

With nested virtualization, these two virtualization mechanisms must themselves be virtualized. In the absence of explicit hardware support for such nested virtualization, host software such as KVM must do this virtualization of the VMCS and EPT in software[11]. Especially since the hardware knows nothing about nesting, only the "bottom-most," Level-0 (L_0) hypervisor (running the ALIBI Observer) uses a native EPT and a native VMCS. The "middle," Level-1 (L_1) hypervisor (the platform's KVM layer in our case) is just a guest of L_0, and so is the nested, Level-2 (L_2) guest holding customer images. This means that the L_0 KVM must maintain a separate VMCS and EPT for its L_1 guest ($VMCS_{01}$ and EPT_{01}), and for its L_2 guest ($VMCS_{02}$ and EPT_{02}). The platform software, L_1, also thinks it is maintaining a VMCS and an EPT for its guest ($VMCS_{12}$ and EPT_{12}).

Nested-virtualization support in KVM allows L_0 to know how L_1 maintains $VMCS_{12}$ and EPT_{12}, and compose them with its own $VMCS_{01}$ and EPT_{01}, to produce appropriate $VMCS_{02}$ and EPT_{02}; doing this efficiently saves unnecessary and costly control transfers across L_0, L_1, and L_2. For VMCSes this is straightforward; L_0 updates its own $VMCS_{02}$ structures according to $VMCS_{12}$ when L_1 issues (and traps on) a `VMWRITE` instruction, and when L_0 passes control to L_2. For EPTs, when L_0 first starts L_2, it marks EPT_{02} empty. Each time that L_2 accesses a nested guest physical address (NGPA) that is not yet mapped in EPT_{02}, an `EPT_VIOLATION` exception occurs, trapping back to L_0, which handles the exception via the `nested_tdp_page_fault` function in KVM; this walks EPT_{12}, trying to find a GPA for the unmapped NGPA; if it finds none, it passes on the job to L_1, by injecting it with the `EPT_VIOLATION` fault; if L_0 does find a mapping in EPT_{12}, it write-protects that mapping (by changing the permissions of the EPT_{01} entry pointing to the page holding the appropriate entry of EPT_{12}), it then adds the mapping to its EPT_{02}, and resumes the L_2 guest.

The write protection of L_1's EPT serves the purpose of monitoring remappings of customer-guest memory by the platform software: if L_1 attempts to modify that mapping in its EPT_{12} – e.g., because it is swapping out a guest physical page – since the memory holding its EPT is write-protected by L_0, an `EPT_VIOLATION` will occur, allowing L_0 to update its EPT_{02} to match the modified mapping by L_1.

6.2 Protected Execution

To offer execution integrity, the Observer at L_0 must protect the contents of the guest (L_2) physical memory, which L_1 maps to L_2, from L_1 itself. L_0 detects allocations by L_1: L_1 marks those allocations in its EPT_{12}, which L_0 monitors, so L_0 is alerted every time such EPT_{12} modifications occur (see Section 6.1). At that time, L_0 write-protects newly allocated pages for as long as L_1 is running. When L_2 starts running, L_0 unprotects those pages, until L_2 exits. Our current prototype does not yet implement vetting of platform-initiated VMCS changes.

6.3 Accounting Case Studies

In addition to the mechanisms ensuring the integrity properties of ALIBI, the prototype addresses the particular case studies we consider as described below.

[11] Several variants exist, but we present here the one we have used, as first described in Turtles [9].

CPU cycles: To measure the CPU cycles used, the Observer takes measurements of the `IA32_TIME_STAMP_COUNTER` model-specific register at each bracketing event: entry into and exit from the instance. The Observer already receives traps for these events with the nested virtualization implementation as described previously in Section 6.1.

To protect the accounting integrity of the timestamp counter, our prototype had to ensure that the register cannot be modified by guests. We do this by enabling an appropriate control field in the related VMCSes ($VMCS_{01}$ for the platform and $VMCS_{02}$ for the customer instance) that causes a trap when the `WRMSR` instruction is executed with the TSC register as an argument. The Observer turns such `WRMSR` instructions into no-ops.

We also take care when the TSC register is set to be virtualized by the platform (this means that the register is auto-loaded from a previously stored value in the VMCS upon entry, and auto-stored back into the VMCS upon exit from that guest). When such virtualization occurs, we measure the advancement of the counter from the virtual value.

Memory: The invariant we maintain for memory accounting is that a customer is charged for a physical page only while that page is accessible to its instance. For the page to be accessible, the EPT_{02} must map it, and the platform (L_1) must have allocated it to the instance.

We record the assignment of ownership of a page to a guest in the Observer when the relevant entry in EPT_{02} is synchronized with EPT_{01} and EPT_{12}. This occurs when a L_2 guest first accesses an assigned page, causing an EPT violation, and L_0 first synchronizes its EPT_{02} entry; and when the L_1 platform modifies a page mapping in EPT_{12}, which causes a protection trap back to L_0. In the latter case, the KVM shadowing logic is used, which marks the relevant entry in EPT_{02} as unsynchronized. Later, when an invalidation occurs (e.g., through the `INVEPT` instruction), L_0 resynchronizes the EPT_{02} entry, unassigning the old page and assigning to the guest the new one (this happens in the `ept_sync_page` function in KVM).

We record the relinquishment of ownership of a page by a guest (i) when a page mapping is modified by L_1 (as described in the previous paragraph), and (ii) when L_1 unmaps a page from a guest, e.g., due to swapping. Then an EPT violation trap to L_0 occurs, and L_0 records the relinquishment.

7. Evaluation

We now present the evaluation of our prototype implementation and analysis of nested virtualization overheads with macro benchmarks that represent real-life CPU/memory and I/O-bound workloads.

Our setup consisted of an HP ML110 machine booted with a single Intel Xeon E31220 3.10GHz core with 8GB of memory. The host OS was Ubuntu 12.04 with a kernel that is based on the KVM git branch "next"[12] with nested virtualization patches[13] added. For both L_1 and L_2 guests we used an Ubuntu 9.04 (Jaunty) guest with the default kernel versions (2.6.18-10). L_1 was configured with 3GB of memory and L_2 was configured with 2GB of memory. For the I/O experiments we used the integrated e1000e 1Gb/s NIC connected via a Netgear gigabit router to an e1000e NIC on another machine.

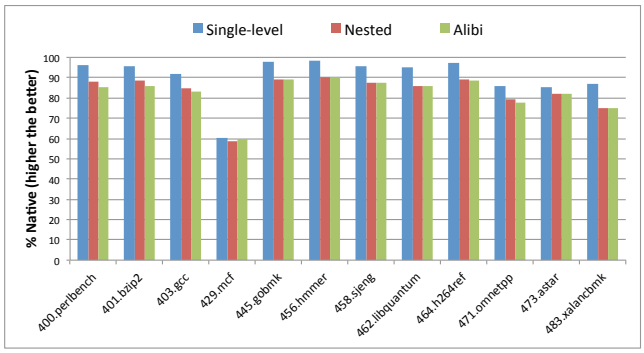

Figure 5. SPEC `CINT2006` results. We see that for most of the CPU intensive benchmarks ALIBI adds little overhead over that of nested virtualization.

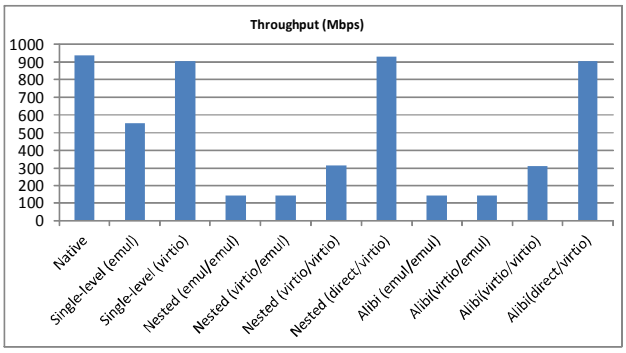

Figure 6. Performance of `netperf` in combinations of I/O virtualization methods between L_0/L_1 and L_1/L_2. `emul`, `virtio` and `direct` refer to device emulation, para-virtualization and direct device assignment, respectively.

7.1 Compute/Memory-bound Workloads

SPEC `CINT2006` is an industry-standard benchmark designed to measure the performance of the CPU and memory subsystem. We executed `CINT2006` in four setups: host (without virtualization), single-level guest, nested guest, and nested guest with ALIBI accounting. We used KVM as both L_0 and L_1 hypervisor with multi-dimensional (EPT) paging. The results are depicted in Figure 5.

We compared the impact of running the workloads in a nested guest (with and without accounting) with running the same workload in a single-level guest, i.e., the overhead added by the additional level of virtualization and accounting. As seen a single-level virtualization imposes, on an average, a 9.5% slowdown when compared to a non-virtualized system. Nested virtualization imposes an additional 6.8% slowdown on average. The primary source of nested virtualization overhead is guest exits due to interrupts and privileged instructions [9] which we expect will diminish with newer hardware [17]. Note that ALIBI's integrity and accounting mechanisms impose a negligibly small overhead ($\approx 0.5\%$) in addition to that imposed by nested virtualization.

We note that this additional overhead imposed by nested virtualization/ALIBI is already quite low given that cloud consumers are willing to pay the cost of single-level virtualization for other benefits such as reduced infrastructure and management costs. We envision verifiable accounting as an *opt-in* service where consumers can choose if they want the additional assurances about accounting; jobs whose owners wish to run without such assurances can be placed by the provider on machines without ALIBI, and the provider can dynamically start machines with or without ALIBI based on demand for the service. Thus, we speculate a $<6\%$ overhead is a small cost that customers may be willing to incur given that it eliminates the (potentially unbounded) uncertainty in accounting that exists today. Given that providers also have an economic and management incentive to motivate adoption of verified accounting, we expect that providers will *subsidize* such services. For example, it is not unreasonable to expect that cloud vendors may offer a 3–6% discount to offset the potential overhead for customers running over ALIBI.

7.2 I/O Intensive Workloads

To examine the performance of a nested guest in the case of I/O intensive workloads we used `netperf`, a TCP streaming application that attempts to maximize the amount of data sent over a single TCP connection. We measured the performance on the sender side, with the default settings of `netperf` (16,384 byte messages).

There are three commonly used approaches to provide I/O services to a guest virtual machine: (i) the hypervisor *emulates* a known device and the guest uses an unmodified driver to interact with it [43]; (ii) a *para-virtual driver* is installed in the guest [36]; or (iii) the hypervisor performs *direct device assignment* where a real device is assigned to the guest which then controls the device directly [17].

These three basic I/O approaches for a single-level guest imply nine possible combinations in the two-level nested guest case. Of the nine potential combinations we evaluate the following interesting cases of virtualization method between L_0/L_1 and L_1/L_2: (a) L_0/L_1 and L_1/L_2 with emulation; (b) L_0/L_1 with para-virtualization and L_1/L_2 with emulation; (c) L_0/L_1 and L_1/L_2 with para-virtualization; and (d) L_0/L_1 with direct device assignment and L_1/L_2 with virtio. Figure 6 shows the results for running the `netperf` TCP stream test on the host, in a single-level guest, and in a nested guest (with and without accounting) using the I/O virtualization combinations described above. We used KVM's default emulated NIC (RTL-8139) and virtio [36] for a para-virtual NIC. All tests used a single CPU core.

On the native system (without virtualization), `netperf` easily achieved line rate (939 Mb/s). Emulation gives a much lower throughput: On a single-level guest we get 60% of the line rate. On the nested guest the throughput is much lower (15% of the line rate) and the overhead is dominated by the cost of device emulation between L_1 and L_2. Each L_2 exit is trapped by L_0 and forwarded to L_1. For each L_2 exit, L_1 then executes multiple privileged instructions, incurring multiple exits back to L_0. In this way the overhead for each L_2 exit is multiplied.

The para-virtual virtio NIC performs better than emulation since it reduces the number of exits. Using virtio for both L_0/L_1 and L_1/L_2 gives 44% of the line rate, better but still considerably below native performance.

Using direct device assignment between L_0 and L_1 and virtio between L_1 and L_2 enables the L_2 guest to achieve near native performance. However, the measured performance is still suboptimal because 70% of the CPU is used for running a workload that takes 48% on the native system. Unfortunately on current x86 architecture, interrupts cannot be directly assigned to guests, so both the interrupt itself and its End-Of-Interrupt (EOI) signaling cause exits. The more interrupts the device generates, the more exits, and

[12] Commit hash `ade38c311a0ad8c32e902fe1d0ae74d0d44bc71e`

[13] The nested virtualization support in KVM is still not mainstream and currently only exists as a patch-set at `http://comments.gmane.org/gmane.comp.emulators.kvm.devel/95395`

therefore the higher the virtualization overhead – which is amplified in the nested case.

The ALIBI CPU and memory accounting overhead in all nested combinations add very little overhead (less than 1%) than what is already imposed by nested virtualization.

Although I/O-bound workload overheads are non-trivial with nested virtualization, we expect recent and future advances in virtualizing or simplifying interrupts [17], as well as (anticipated) hardware support for nested virtualization [37] to reduce this overhead significantly.

8. Discussion

TCB size: We acknowledge that in our current implementation, ALIBI does not meet our goal of having a minimal trusted base. Since ALIBI relies on the nested virtualization support in KVM, it has to invariably include the KVM codebase and the Linux kernel itself in its TCB. This is an artifact of our current prototype and our pragmatic choice in choosing KVM because of the readily available codebase and nested virtualization support. The actual protection mechanisms that ALIBI adds are negligible (few hundred lines of code) over the basic nested virtualization support. We believe that this can be added to more lightweight nested virtualization solutions including CloudVisor [48] and XMHF [45].

Stochastic correctness: It is possible to provide a weaker form of accounting integrity without explicitly providing image and execution integrity. This might make sense under a weaker threat model where the customer is running on a benign, bug-free platform. In this case, "good faith" usage observations might give loose assurances to customers by external randomized auditing mechanisms. For example, the customer can create a known workload with a prespecified billing footprint and synthetically inject it into the cloud to see if there are obvious discrepancies. In our threat model, the platform may inflate costs or may have bugs exploitable by others. In this case, this form of "stochastic accounting integrity" without execution and image integrity is less applicable as it tells the customer nothing about what code actually incurred the charges.

Multi-core support: Our current prototype implementation supports a single processor core. We chose to support only a single core primarily for ease of debugging. There is no fundamental limitation either in the Linux-KVM codebase or in the ALIBI architecture that precludes support for multi-core platforms. For example, Linux-KVM nested virtualization already maintains multiple VMCS/VCPU/EPT structures for SMP support. Our existing prototype can be reinforced with SMP support by simply converting the existing global data structures for CPU and memory accounting and memory protection logic to be VCPU-relative. We also note that some of the current best practices in public clouds make support for multi-core much easier. Although we have yet to implement such support, we are cautiously optimistic.

Resources expended by providers: Our design does not currently account for external costs that a provider or ALIBI incurs on behalf of a specific customer: e.g., cycles for servicing hypercalls, or due to cache/memory contention. These costs can be ameliorated by better job placement, so the platform should be in part responsible. Another alternative is that these costs may be amortized into the billing mechanism if the provider can have an estimate of the overhead it incurs as a function of the offered load. We are also considering more systematic causality-based tracking to attribute system/Alibi costs to the proper job, to enable different charging models.

Physical attacks: The root of trust in ALIBI lies with the TPM chip on the provider's infrastructure. If the provider can physically tamper with properties of the TPM chip, she can tamper with the integrity of the Observer without being detected by customers, which can, in turn, turn a blind observing eye to provider tampering with the verifiable-accounting properties of ALIBI. Although extremely difficult, attacks against TPM properties have been demonstrated – for example, via *cold-boot attacks* [19] that recover from memory TPM encryption or signing keys, or more sophisticated hardware-probing attacks. Such attacks are in the purview of a sophisticated platform today, but will reduce in feasibility as trusted-execution functionality moves deeper into the hardware platform. For example, an MMU that directly encrypts memory never puts secret data such as keys on DRAM and, therefore, eliminates cold-boot or bus eavesdropping attacks. Today's TPM chips, although not tamper-resistant, are tamper-evident: physical attack against them renders them visibly altered. Periodic physical inspection by an external compliance agency, akin to a Privacy CA, might be a plausible interim solution. What is more, CPU manufacturers hint that trusted execution without an external TPM chip might be coming in their future products [47]; physically attacking CPUs is significantly harder than attacking motherboard-soldered chips.

9. Related Work

We discuss related work in different aspects of cloud computing and trusted computing and place these in the context of our work for enabling verifiable resource accounting.

Nested virtualization: While the idea of nested virtualization has been around since the early days of virtualization, it is only recently that we see practical implementations. The two works closest to ALIBI in this respect are Turtles [9] and CloudVisor [48]. ALIBI builds on and extends the memory protection techniques that these approaches develop. The key difference, however, is in the applications and threats that these systems target. Turtles is focused on being able to run any hypervisor in the cloud and other security properties (e.g., to protect against hypervisor-level rootkits). CloudVisor, on the other hand, is designed to prevent a malicious platform operator from inferring private information residing in a guest VM's memory. These systems differ in one key aspect: CloudVisor does not attempt to provide a full-fledged hypervisor for multi-level nested virtualization that Turtles can provide. In this respect, ALIBI is arguably closer to CloudVisor in that we only need one more level of virtualization and do not need multi-level nesting. At the same time, however, some of the mechanisms in CloudVisor (e.g., encrypting pages) are likely an overkill for ALIBI, since we only care about integrity and not confidentiality. CloudVisor further assumes that the cloud provider has no incentive to be malicious or misconfigured, which is not true in the accounting scenarios we tackle. Consequently, it does not provide any correctness of the accounting and execution integrity properties. That said, the ALIBI extensions can be easily added to the CloudVisor implementation as well if the sources are made available.

Attacks in the cloud: The multiplexed and untrusted nature of cloud environments leads to attacks by co-resident tenants and by the providers themselves. These include side channel attacks to expose confidential information or identify co-resident tenants [35, 49]. More directly related, there are practically demonstrated attacks against today's cloud accounting including attacks against management interfaces [28, 42] and current resource management mechanisms [44, 50]. Liu and Ding discuss a taxonomy of potential attacks [27]. Our goal is to protect against these specific types of accounting vulnerabilities and at the same time allow cloud providers to be able to justify the resource consumption.

Cloud accountability: Cloud customers may want to ensure that the provider faithfully runs their application software and respects input-output equivalence [18]; that has not tampered or lost their data [8, 23]; and respects certain performance SLAs [32]. These target other types of accountability; our work focuses specifically on trustworthy resource accounting.

Cloud monitoring and benchmarking: Recent work from Li et al. compares the costs of running applications under different popular providers [26]. Other work makes the case for a unified set of benchmarks to evaluate cloud providers [11, 21]. Several efforts have identified challenges in scalably monitoring resource consumption in cloud and virtualized environments [6, 14, 33]. While such tools are also motivated by resource monitoring, they do not focus on verifiability of the measurements.

Integrity: There is rich literature on protecting control flow integrity [15]. Such work guarantees that a program follows valid execution paths allowed by the control flow graph. While this guarantee is necessary for accounting correctness, it is not sufficient. For example, without the protections we enable, the provider could arbitrarily inflate the resource footprint by forcing the program to take valid but unnecessary code paths. Image integrity is related to the recent work on Root of Trust for Installation [38] but in the cloud context.

Rearchitecting OS and hypervisors: As we discussed earlier, one could envision clean-slate solutions where the operating system and the hypervisor are rearchitected to support resource accounting as a first-class primitive and also minimize the threat surface. This includes recent work revisiting the design and implementation of isolation kernels [25] and other work on microkernel-like hypervisors [13]. By leveraging nested virtualization, our work explores a different point in the design space and incurs a small overhead in favor of immediate deployability.

10. Conclusions and Future Work

As computation is rapidly turning into a "utility," the need for trustworthy metering of usage is ever more imminent. The *multiplexed* and *untrusted* nature of cloud computing makes the problem of accounting not only more relevant but also significantly more challenging compared to traditional utilities (e.g., water, power, ISPs). For example, providers may have incentives to be malicious to increase their revenues; other co-resident or remote customers may try to steal resources for their own benefit; and customers have obvious incentives to dispute usage. What is fundamentally lacking today is a basis for verifiable resource accounting leading to severe sources of uncertainty and inefficiency for all entities involved in the cloud ecosystem.

As a first step to bridge this gap, we present the design and implementation of ALIBI. Our design reflects a conscious choice to enable cloud customers and providers to benefit from ALIBI with minimal changes. To this end, we envision a novel, and perhaps viable, use-case for nested virtualization. We demonstrate practical protection schemes against a range of existing accounting vulnerabilities. Our implementation adds negligible overhead over the cost of nested virtualization; we expect that future hardware and software optimizations will further drive these overheads down, in the same way that the adoption of cloud computing spurred innovation in traditional virtualization technologies.

We acknowledge the need to address a range of additional concerns to realize the full vision of verifiable accounting. This includes the need for better formalisms to reason about accounting equivalence, accounting for I/O resources, carefully attributing provider-incurred cost (e.g., cost of hypercalls, cost of power/cooling), among other factors. While we fully expect to run into significant "brick walls" in addressing these issues, the initial success shown here, the experiences we gained in the process, and emerging processor roadmaps give us reasons to be optimistic in our quest.

Acknowledgments

We thank our shepherd, Gernot Heiser, for his help while preparing the final version of this paper, as well as the anonymous reviewers for their detailed comments. Rekha Bachwani, Yanlin Li, John Manferdelli, and David Wagner have provided valuable ideas and feedback. Nadav Har'El helpfully answered our questions about the pending Turtles nested-virtualization optimizations in the mainline Linux-KVM codebase. This work was funded in part by the Intel Science and Technology Center for Secure Computing.

References

[1] Cloud storage providers need sharper billing metrics. `http://www.networkworld.com/news/2011/061711-cloud-storage-providers-need-sharper.html?page=2`.

[2] dm-verity: device-mapper block integrity checking target. `http://code.google.com/p/cryptsetup/wiki/DMVerity`. Retrieved 2/2013.

[3] IT Cloud Services User Survey: Top Benefits and Challenges. `http://blogs.idc.com/ie/?p=210`.

[4] Service billing is hard. `http://perspectives.mvdirona.com/2009/02/16/ServiceBillingIsHard.aspx`.

[5] TPM Main Specification Level 2 Version 1.2, Revision 103 (Trusted Computing Group). `http://www.trustedcomputinggroup.org/resources/tpm_main_specification/`.

[6] VMWare vCenter Chargeback. `http://www.vmware.com/products/vcenter-chargeback/overview.html`.

[7] The Trusted Boot Project (tboot). `http://tboot.sourceforge.net/`, Sept. 2007.

[8] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable Data Possession at Untrusted Stores. In *ACM CCS*, 2007.

[9] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *OSDI*, 2010.

[10] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks are Realistic Threats. In *USENIX Security*, 2005.

[11] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *Proc. MASCOTS*, 2011.

[12] R. Cohen. Navigating the Fog - Billing, Metering and Measuring the Cloud. Cloud computing journal `http://cloudcomputing.sys-con.com/node/858723`.

[13] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *SOSP*, 2011.

[14] J. Du, N. Sherawat, and W. Zwaenepoel. Performance Profiling in a Virtualized Environment. In *Proc. HotCloud*, 2010.

[15] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *OSDI*, 2006.

[16] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and Secure Distributed Read-only File System. *ACM TOCS*, 20(1), 2002.

[17] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. ELI: Bare-Metal Performance for I/O Virtualization. In *ASPLOS*, 2012.

[18] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable Virtual Machines. In *OSDI*, 2010.

[19] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest

We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX Security*, 2008.

[20] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring Operating System Kernel Integrity with OSck. In *ASPLOS*, 2011.

[21] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Proc. ICDE Workshops*, 2010.

[22] R. Iyer, R. Illikkal, L. Zhao, D. Newell, and J. Moses. Virtual Platform Architectures for Resource Metering in Datacenters. In *SIGMETRICS*, 2009.

[23] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *ACM CCS*, 2007.

[24] B. Kauer. OSLO: Improving the Security of Trusted Computing. In *USENIX Security*, 2007.

[25] A. Kvalnes, D. Johansen, R. van Renesse, F. B. Schneider, and S. V. Valvag. Design Principles for Isolation Kernels. Technical Report 2011-70, Computer Science Department, University of Tromsø, 2011.

[26] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *IMC*, 2010.

[27] M. Liu and X. Ding. On Trustworthiness of CPU Usage Metering and Accounting. In *ICDCS-SPCC*, 2010.

[28] M. McIntosh and P. Austel. XML signature Element Wrapping Attacks and Countermeasures. In *ACM SWS*, 2005.

[29] A. Mihoob, C. Molina-Jimenez, and S. Shrivastava. A Case for Consumer-centric Resource Accounting Models. In *Proc. International Conference on Cloud Computing*, 2010.

[30] J. C. Mogul. Operating systems should support business change. In *HotOS*, 2005.

[31] B. Parno. Bootstrapping Trust in a "Trusted" Platform. In *HotSec*, 2008.

[32] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proc. USENIX ATC*, 2011.

[33] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro*, 2010.

[34] K. Ren, C. Wang, and Q. Wang. Security Challenges for the Public Cloud. *IEEE Internet Computing*, 16(1), 2012.

[35] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get off of my cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *ACM CCS*, 2009.

[36] R. Russell. virtio: Towards a De-Facto Standard for Virtual I/O Devices. *ACM SIGOPS OSR*, 42(5), 2008.

[37] R. Sahita. Intel Virtualization Technology Extensions for High Performance Protection Domains. `https://intel.activeevents.com/sf12/scheduler/catalog.do`, Sept. 2012. Intel Developer Forum 2012, Session ID FUTS003.

[38] J. Schiffman, T. Moyer, T. Jaeger, and P. McDaniel. Network-Based Root of Trust for Installation. *IEEE Security and Privacy*, 9(1), 2011.

[39] V. Sekar and P. Maniatis. Verifiable Resource Accounting for Cloud Computing Services. In *ACM CCSW*, 2011.

[40] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP*, 2007.

[41] M. A. Shah, M. Baker, J. C. Mogul, and R. Swaminathan. Auditing to Keep Online Storage Services Honest. In *HotOS*, 2007.

[42] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono. All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces. In *ACM CCSW*, 2011.

[43] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *USENIX ATC*, 2001.

[44] V. Varadarajan, B. Farley, T. Ristenpart, and M. M. Swift. Resource-Freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense). In *ACM CCS*, 2012.

[45] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *IEEE S&P*, 2013.

[46] M. Wachs, L. Xu, A. Kanevsky, and G. R. Ganger. Exertion-based Billing for Cloud Storage Access. In *HotCloud*, 2011.

[47] A. Wolfe. Intel CTO Envisions On-Chip Data Centers. `http://www.informationweek.com/news/global-cio/interviews/showArticle.jhtml?articleID=221900325`, Nov. 2009.

[48] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *SOSP*, 2011.

[49] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM CCS*, 2012.

[50] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler Vulnerabilities and Coordinated Attacks in Cloud Computing. In *IEEE NCA*, 2011.