Configuring Cloud Deployments for Integrity

Trent Jaeger, Nirupama Talele, Yuqiong Sun, Divya Muthukumaran,
Hayawardh Vijayakumar, and Joshua Schiffman
{tjaeger,nrt123,yus138,muthukumaran,hvivay,jschiffm}@cse.psu.edu

Systems and Internet Infrastructure Security Lab
Pennsylvania State University

Abstract. Many cloud vendors now provide pre-configured OS distributions and network firewall policies to simplify deployment for customers. However, even with this help, customers have little insight into the possible attack paths that adversaries may use to compromise the integrity of their computations on the cloud. In this paper, we leverage the pre-configured security policies for cloud instances to compute the integrity protection required to protect cloud deployments. In particular, we show that it is possible to compute security configurations for cloud instance deployments that can prevent information flow integrity errors and that these configurations can be measured into attestations using trusted computing hardware. We apply these proposed methods to the OpenStack cloud platform, showing how web server application instance can be configured to protect their integrity in the cloud and how integrity measurement can be used to validate such configurations for approximately 3% overhead.

1 Introduction

Cloud computing is a realization of computing as a utility, where customers submit their computing tasks to a centralized service that provides the resources necessary to execute those tasks. According to NIST [33], "[c]loud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources." Rather than purchasing and maintaining an abundance of hardware resources themselves, customers can "plug in" to the cloud, paying for only the quantity of resources used. This is particularly attractive to those customers whose resource utilization may vary dramatically or where the costs of hardware and its maintenance form a significant fraction of their overall budget.

Despite a promising business model, security is a major concern that may limit the impact of the cloud computing paradigm. Customers need assurance that their personal or proprietary processing can be protected on systems administered by a third party. For example, if a medical billing organization wants to use a cloud system to run their processing, they will have to consider how to protect their processing from remote adversaries and achieve compliance with privacy requirements, such as HIPAA, when such computing is moved to a third party. Key to the safe use of any computing system is the ability to configure security policies that limit adversary access to critical processing. While the cloud vendors aim to make security configuration simpler, through pre-configured OS distributions and default firewall policies, significant responsibility for security decisions still falls upon the customers [18].

At Penn State's Systems and Internet Infrastructure Security (SIIS) Lab, we are studying how cloud vendors and customers can work together to configure cloud computations that protect customer processing. This study spans two critical issues in cloud computing: (1) configuring cloud computations to prevent known attacks on such pre-configured OS distributions and (2) validating the runtime compliance of cloud computations with the expected configurations.

First, in modern systems, much of the security configuration is already done in advance. Now, administrators configure hosts by selecting an OS distribution, selecting the desired application programs, and

configuring network access for the deployed distribution. Selecting a commodity OS distribution now often implies selecting a mandatory access control (MAC) policy to be enforced by the distribution [53, 59, 36, 35, 28], which limits the number of processes accessible to remote adversaries and aims to confine the rest. Due to the complexity of MAC policies, administrators do not modify such policies manually, limiting the custom defenses that they can apply to the selection of software packages and configuration of firewall rules. Further, such policies are designed for *least privilege* [41], meaning that functionality drives which permissions are included, not security concerns. As a result, the MAC policy may not accurately enforce the integrity requirements that the administrators may expect, in such cases permitting operations that would compromise their integrity requirements (i.e., if the administrators understood the MAC policy). With the advent of cloud computing, the limitations of this approach to configuration has been transferred into a problem for cloud customers.

Second, once the customer has settled on a configuration for their cloud instance, they want to know whether the runtime execution of their instance will behave as expected. Trusted computing mechanisms have been developed to collect measurements of system events necessary to produce proofs of system integrity (attestations) that can be verified by remote parties [39, 49, 16, 50, 51, 44, 25, 24, 5]. However, trusted computing has not been widely adopted on traditional hosts, and cloud computing provides additional challenges because the cloud is opaque to customers. For example, the node controllers upon which instances are deployed may not be addressable by remote customers and cloud instances may be migrated dynamically.

In this paper, we describe methods for: (1) identifying and resolving integrity problems in the security policies resulting from the deployment of cloud computations using pre-configured OS distributions and (2) measuring and validating that the runtime integrity of deployed systems complies with the integrity-protecting configuration. This work leverages several research projects underway at the SIIS Lab. First, we demonstrate that we can use available package configurations and MAC policies in OS distributions to locate individual program entrypoints that are accessible to adversaries [56], called the *attack surface* of the program [15]. Second, using this information, we can configure information flow problems whose solutions mediate adversary access, blocking potential attacks paths [29]. Such problems can be constructed to evaluate remote or local threats against individual cloud instances or threats against a computation consisting of several instances. Third, we describe methods for enforcing such mediation in operating systems [55] and programs [47, 21, 31], enabling customers to deploy cloud instances that protect their integrity proactively. Finally, we describe our trusted computing mechanism for measuring cloud instances, based on an *integrity verification proxy* (IVP) service [46, 45], which monitors customer integrity criteria on VMs running in the cloud.

We demonstrate these methods for Ubuntu Linux OS distributions using SELinux deployed on the OpenStack cloud platform. What we find is that: (1) we can produce policies for cloud instances that approximate classical integrity, in the form of Clark-Wilson integrity [10] and (2) we can monitor the enforcement of such policies using trusted computing with low overhead. Using such techniques, cloud vendors can provide customers with insight into how customization may impact the integrity of their deployments and monitor the execution of their deployments.

The remainder of this paper is as follows. Section 2 provides background on Infrastructure as a Service (IaaS) cloud platforms, which are the target of this work. Section 3 defines the information flow integrity problem that we aim to address in this paper. Section 4 describes our approach for configuring cloud computations for integrity. Section 5 outlines how we monitor cloud computations for their adherence to that configuration. Section 6 concludes and discusses future work.

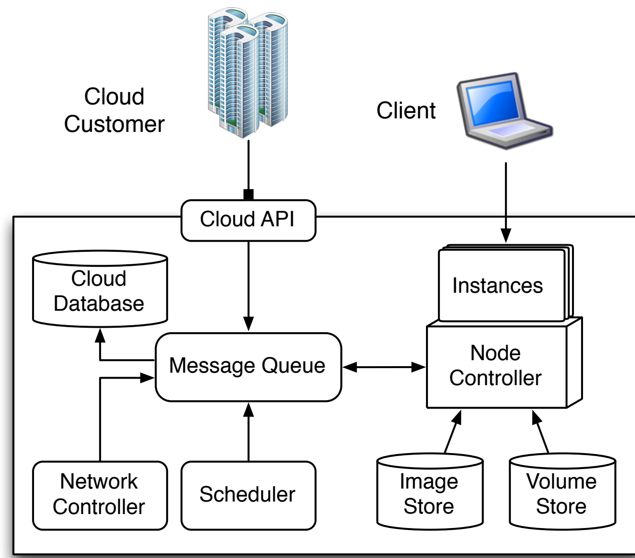

Fig. 1. IaaS Clouds.

2 IaaS Cloud Architecture

Clouds come in a variety of architectures with differing levels of service and features. While the definition of a "cloud" is as nebulous as its name, NIST has begun to categorize clouds based on the degree of administration and services the clouds offer to the customer [33]. Cloud computing provides a platform for customers to run *instances*, computations containing customer-chosen software and data. Cloud platforms offer different granularity of processing, such as Software as a Service [40, 14] (SaaS), Platform as a Service [26, 13] (PaaS), and Infrastructure as a Service [4, 1] (IaaS). In this paper, we consider only IaaS clouds as they are the building blocks of higher level cloud abstractions. Moreover, the IaaS paradigm gives the customer more control over how cloud components manage sensitive data and code. IaaS clouds provide the basic infrastructure launch instances in the form of virtual machines (VMs), such as VM hosting, virtualized networking, and storage for VM images and disk volumes. Customers can use IaaS clouds to replace or supplement a traditional data center by hosting the service in the cloud and scaling up compute and storage requirements on-demand.

As an illustrative example, consider the high-level IaaS cloud architecture in Figure 1. The primary component is the *node controller*, a VM host for customer VM instances. Clouds are composed of thousands of these nodes, which are broken into clusters that provide a level of redundancy and can be spread out geographically based on demand of the region. Within each cluster, a *network controller* is responsible for configuring virtual networking between instances and translating public IPs to private intra-cloud addresses. When a new instance is requested, a *scheduler* chooses a node controller to host the VM based on various scheduling policies like resource fairness. In addition, the cloud's state (e.g., which instances are running) is stored in a *cloud database* that is updated and referenced by the various components.

Instances are created from disk images stored in the *image store*. They are uploaded to the cloud by the customer or a third party vendor and remain static across reboots. Additional mutable storage is provided

through additional services like a key-object stores (e.g., Amazon's S3 [3]) or network attached block storage from a *volume store*. Customers control their instances through an API endpoint, which also exposes options for configuring firewall rules, `ssh` host identity keys, and other policies. Finally, instances open to the internet can interact with clients to provide services for which they were designed.

The cloud instances that we deploy are Ubuntu Linux OS distributions that run a particular application package, such as a web server (Apache), database (MySQL), or web client (Firefox). Each of the distributions we use includes an SELinux mandatory access control (MAC) policy [36] that governs the accesses of the running processes. SELinux policies are composed from individual policies designed for software packages. The SELinux policies are produced using a runtime analysis, which biases them toward *least privilege* [41]. That is, SELinux policies restrict processes based on the functionality required for the associated program to run, not based on protecting the integrity of the process. As a result, adversaries often have access to some of the resources used by processes, but there is no principled approach to identify those cases and protect the process. Other commodity MAC enforcement works similarly [35, 59, 53, 28].

In discussing cloud integrity, we assume the physical security of the cloud is maintained and that attacks on the hardware are prevented. We also trust the cloud at an organizational level to provide services without malicious intent. That is, we assume the cloud's components were honestly configured with the purpose of protecting the integrity and confidentiality of its customers. However, we do not trust the cloud beyond that point and accept that curious or malfeasant administrators may attempt to alter cloud systems in an untrustworthy way. Thus, we consider threats like an administrator logging in to the node controller to directly read instance memory or locally cached disk images. We also consider threats from network attackers both within the cloud intranet and externally that can snoop on, alter, or inject packets. We do not guarantee that satisfaction of an integrity criteria implies that the system will not perform undesirable behavior. We leave it up to the relying party to design integrity criteria that would ensure this property.

3 Information Flow Integrity Problem

To detect all possible integrity threats, we must identify when an adversary can write to data that may be read or executed the victim [6], which can be modeled as an information flow problem. Traditionally, an information flow problem is defined as follows:

Definition 01 *An* information flow problem, $\mathcal{I} = (\mathcal{G}, \mathcal{L}, \mathcal{M})$*, consists of the following concepts:*

1. *A* directed data flow graph $G = (V, E)$ *consisting of a set of nodes* V *connected by edges* E*.*
2. *A* lattice $\mathcal{L}$= $\{L, \preceq\}$*. For any two levels* $l_i, l_j \in L$*,* $l_i \preceq l_j$ *means that* l_i *'can flow to'* l_j*.*
3. *There is a* level mapping function $M : V \rightarrow \mathcal{P}^L$ *where* $\mathcal{P}^L$ *is the power set of L (i.e., each node is mapped either to a set of levels in L or to* $\emptyset$*).*
4. *The lattice imposes security constraints on the information flows enabled by the data flow graph. Each pair* $u, v \in V s.t.\ [u \hookrightarrow_G v \wedge (\exists l_u \in M(u), l_v \in M(v).\ \ l_u \not\preceq_{\mathcal{L}} l_v)]$*, where* $\hookrightarrow_G$ *means there is a path from* u *to* v *in* G*, represents an* information flow error.

Such an information flow problem can be constructed for mandatory access control (MAC) policies. The data flow graph is determined by the information flow (i.e., read and write operations) authorized by the MAC policy. The integrity lattice identifies the types of security-sensitive entities, such as integrity-critical resources and adversary-accessible resources. The mapping assigns the relevant levels in the lattice to the MAC policy labels for those entities. It has been shown that information flow errors in programs [30] and MAC policies [17, 42, 9] can be automatically found using such a model.

However, resolving such information flow errors has been a complex manual task. In general, information flow integrity errors can be resolved by changing the data flow graph (i.e., the MAC policy) or adding *mediation* to change the integrity of data propagated by information flows. However, changing the data flow graph is difficult in practice because it implies a change in the operations a system may perform, which may prevent one or more programs from functioning correctly. As a result, we explore methods to to resolve information flow errors using mediation.

The challenge in this work is two-fold. First, we must map the configuration of cloud components to a system-wide information flow problem and determine which mediation placements are viable. Prior work examines information flows among VMs in a cloud environment [7], but does not examine problems caused by such flows within the cloud components themselves. Second, we want enable customers to verify that their running instances only receive untrusted inputs via mediators. We leverage trusted computing technologies [54], but we must adapt this work to cloud computing in a manner that enables verification of the guarantee above at runtime.

4 Configuring for Integrity

In this section, we explore a method for configuring cloud instances and computations consisting of multiple cloud instances to protect their integrity. As described above, each cloud instance is a pre-configured OS distribution, but the security policies in each distribution are not designed to protect integrity. However, these distributions do include information sufficient to compute adversary accessibility to program entrypoints (see Section 4.1) from which threats to cloud computation deployments can be identified (see Section 4.2). We then examine defenses for these threats that would provide proactive integrity protection system-wide in Section 4.3.

4.1 Computing Attack Surfaces

Researchers have explored several methods to describe how adversaries may use their access rights to launch attacks. For example, methods have been developed to compute *attack graphs* [48, 37, 34], which generate a sequence of adversary actions that may result in host compromise. However, these methods treat programs as black boxes, where rules describe possible compromises without principled information about the programs. Should a particular vulnerabilities be patched in the program code, then attack graphs may diverge from the actual deployment, leading to false positives.

As an alternative, researchers have argued for defenses at a program's *attack surface* [15], which is defined by the entry points of the program accessible to adversaries. In practice, a program entrypoint is an instruction in the program that invokes the system call library to receive system resources (e.g., files, network, or IPC data). Unfortunately, programs often have a large number of entrypoints, and it is difficult to know which of these are accessible to adversaries using the program alone. Some experiments have estimated attack surfaces using the value of the resources behind entrypoints [23]. However, if the goal is simply to take control of a process, any entrypoint may suffice. While researchers have previously identified that both the program and the system security policy may impact the attack surface definition [15], methods to compute the accessibility of entrypoints had not been developed.

In a recent paper, we develop a method to compute program attack surfaces for OS distributions [56]. Calculating the attack surface has two steps. First, for a particular subject (e.g., the SELinux label `httpd_t`), we need to define its adversaries (e.g., processes with the SELinux subject label `user_t`), and locate OS objects under adversarial control (e.g., files with the SELinux object label `httpd_user_content_t`). We do this using the system's MAC policy. Next, we need to identify the program entry points that access

these adversary-controlled objects. Statically analyzing the program cannot tell which permissions are exercised and which OS objects accessed at each entry point, and thus we use a runtime analysis to locate such entry points.

The adversaries of a subject are identified conservatively. For a particular subject in the MAC policy, the only other subjects that it trusts are those that have permission in the MAC policy to modify its executable program file (directly or indirectly). All other subjects are untrusted. A detailed method is specified [56] to identify all labels accessible to subject that may be modified by adversaries.

Practical runtime analysis is possible for many programs because several Linux software packages now include comprehensive test suites. These test suites test program functionality across multiple configuration options, which often identifies attack surfaces that would not be found through normal manual execution. Despite the conservative definition for adversaries, our study showed that only a small fraction of the program entrypoints are actually accessible to adversaries. For 23 system subjects in the Ubuntu Linux 10.04 Desktop distribution, only 81 out of 2138 entrypoints are accessible to adversaries. For well-known server programs, 14 out of 78 entrypoints in OpenSSH and 5 out of 30 entrypoints for Apache were accessible to adversaries. Using knowledge of these entrypoints, we identified two previously-unknown vulnerabilities, demonstrating the importance of tracking attack surfaces to prevent vulnerabilities. As a result, we found that computing attack surfaces is possible, yields useful information for extending proactive integrity protections, and reduces the effort of defenders greatly in determining where to provide integrity protections.

4.2 Computing System-Wide Mediation

With knowledge of how to compute attack surfaces[1], which identify adversary accessibility, our next goal is to compute the mediation mediators necessary to resolve all information flow errors in a cloud computation. Researchers have found that it is possible to find resolution to an information flow errors by computing a solution to a graph-cut problem [19, 20, 38]. In this case, the graph-cut solution corresponds to the placement of mediation code necessary to resolve all information flow errors. This solution applies for a two-level integrity lattice (e.g., high and low integrity), but in practice, more than two integrity requirements may be necessary resulting in a *multiway cut problem*, which has been shown to be NP-Hard for directed graphs [12]. However, greedy solutions are generally effective for finding possible mediations.

The problem is how to use this knowledge to configure integrity protections for cloud computations, possibly spanning multiple cloud instances. Using available security policies (i.e., MAC and firewall policies), it is possible to produce an information flow problem, as defined in Definition 01 above. However, often over 100 mediators are required per instance. This seems like to much work for customers, programmers, administrators, or OS distributors.

In a paper to appear in December 2012 [29], we use the insight that pre-configured instances face several threats in any deployment of that instance. As a result, these threats should always be mediated proactively, and the focus should be on the new threats that emerge in the specific customer deployment. A customer deployment may impact the cloud instance in two ways: (1) it may add new remote threats by expanding the network connections to any instance and (2) it may add local threats through the introduction of unprivileged code and unverified data to the instance. Thus, we solve information flow problems that identify the program entrypoints necessary to protect the instance integrity by default, from remote threats given the deployment, and from local threats given the deployment.

[1] In particular, how to compute which program entrypoints access which object labels in the MAC policy. This computation does not use the same threat model as the attack surface calculation, but rather adversaries are based on the information flow problem lattice in Definition 01.

For an Apache web server cloud instance, no more than 217 mediators are necessary to protect all the subjects in a default network configuration. These should be defended by default, otherwise all uses of this OS distribution would be flawed. In practice, cloud vendors and OS distributors should work together to ensure proactive integrity protection for these entrypoints.

When a web application is deployed on this server, 24 additional entrypoints must mediate remote threats, 10 of which are specific to the new application code. Thus, to protect the integrity of the customer's instance from new remote threats caused by the deployment, the customer must determine whether the additional program entrypoints accessible to adversaries are protected. In this case, the customer must check their own code for mediation as well as obtain insight into mediation for 14 new attack surface entrypoints in existing OS distribution code. At present, no automated approach exists to resolve these issues, but hopefully, the identification of attack surfaces will motivate such methods.

In addition, we also examine attack surfaces that may result from local threats. In this experiment, we identify local threats as the untrusted object labels relative to the web server process that are not reachable from remote adversary input. If each of these object labels do indeed include malicious data, then 56 additional mediators are necessary to protect the web server and trusted computing base processes. This insight is not altogether surprising, as local exploits are a common vector for launching attacks. In this case, it behooves the customer to ensure that any data assigned to these untrusted object labels is vetted prior to deploying the cloud instance.

Finally, our method also computes the mediation required for a cloud computation consisting of multiple cloud instances. We configured a web application that included a database as well as two distinctly-configured web clients. In addition, we also evaluated the mediation required in the node controller hosts of the cloud instances[2], resulting five VMs total. An advantage is that the same mediation requirement may be present across the system at large, where 2/3 of the attack surface entrypoints appeared in multiple VMs. As a result, 525 mediators total are needed for the combination of VMs by default. Redundant mediation also helped limit the impact of dealing with remote threats, as only three new attack surface entrypoints required mediation once the customer deployment was configured. Local threats seem to be more dependent on the application being hosted on the VMs, however. The new attack surface resulting from local threats on web clients had little overlap with that of the web server, meaning that addressing local threats will be an important challenge in the future.

4.3 Approximating Clark-Wilson Integrity

Once mediation locations have been identified, enforcement code that implements that mediation is necessary. Enforcement code may be added to the operating system or the program to mediate attack surfaces. We describe the two cases and their impact relative to achieving Clark-Wilson integrity, a classical integrity model.

In the first case, the operating system provides access control to limit access to processes, but as we have shown that available access control does not prevent attack surfaces from appearing. The operating system can do two additional things to limit attack surfaces. First, the operating system can limit the entrypoints that processes can use to access resources controlled by adversaries. That is, operating systems can prevent processes from expanding their attack surface beyond what is known. Second, when a process requests a resource from the operating system by name, the operating system can limit the adversaries' abilities to redirect that name resolution. For example, when a process requests a resource by pathname, if an adversary can modify a directory used in name resolution then an adversary may redirect the victim to a resource of the adversary's choosing. In a recent study, we found that many latent vulnerabilities to this threat exist in

[2] In this case, we used a Xen-based system.

programs, even mature ones [58]. To control access to the program entrypoints, we have built a *process firewall* mechanism that can enforce rules to protect entrypoints from adversary access in name resolution and to the returned resource [57].

In the second case, a process entrypoint expects to receive some untrusted input. In this case, it is up to the program to protect itself from such inputs. In recent work, a variety of mechanisms have been explored to enforce type safety [32, 11], enforce execution semantics [2, 8], enforce information flow [31], and use capabilities to control interaction with adversaries [22, 21, 47]. At present, program defenses of attack surfaces are ad hoc in practice, but methods such as these and others may greatly limit the choices that adversaries can make that may impact process integrity.

Protecting program entrypoints from adversaries is an approximation of Clark-Wilson integrity [10]. In this approximation, the Clark-Wilson requirement that all programs that process high integrity data be certified (i.e., formally-assured) is dropped (rule C2), so the Clark-Wilson requirement that all untrusted inputs to such processes must be discarded or upgraded (C5) is strengthened to only allow untrusted inputs to mediating entrypoints. That is, the set of mediating entrypoints must be a superset of the attack surface entrypoints. That is the ultimate goal of this work.

5 Monitoring Computation Integrity

In this section, we describe a method for monitoring the runtime integrity of cloud computations. Because we have carefully evaluated that the security configuration provides the required integrity protection, as described in the previous section, the goal is to monitor that the cloud computation is enforcing that configuration. Monitoring a configuration is much more efficient than monitoring memory in general, as configurations change infrequently, if at all.

5.1 Measuring Integrity

Validating trust that a remote system will behave as expected is a fundamental problem in computer security. The introduction of trusted computing hardware, in particular the Trusted Platform Module [54] (TPM), has been one of the most significant advances to solve this problem in recent years. TPM devices provide protected storage for collecting measurements of host integrity (integrity measurements) and cryptographic mechanisms suitable for constructing proofs from integrity measurements that can be verified by remote parties (attestations). Despite the broad availability of TPMs, in over 200 million hosts, and several trusted computing mechanisms [49, 51, 50, 39, 25] hardware-based trusted computing has not yet led to the widespread adoption of mechanisms to validate trust in commodity systems. We find that such usage has been limited by: (1) the difficulty in predicting what "integrity" means for complex commodity system and (2) the lack of support for runtime integrity measurement. In this research thrust, our goals have been to design new integrity measurement mechanisms based on the TPM hardware to support practical integrity measurement for commodity systems and apply these techniques to different kinds of system deployments.

We have developed a number of integrity measurement designs over the years to capture more events that may impact integrity and ease the verifiers' task [52, 27, 44], and our most recent work unifies these ideas [46]. The key concept in this paper is the *integrity verification proxy* (IVP), an integrity monitor framework that verifies system integrity at the proving system on behalf of the remote clients (e.g., cloud customers). The IVP is a service resident in a virtual machine (VM) host that monitors the integrity of its hosted VMs for the duration of their execution through a combination of load-time and VM introspection mechanisms. Client connections to the monitored VM are proxied through IVP and are maintained so long as the VM satisfies client-supplied integrity criteria. Runtime VM introspection can be costly, but we show that

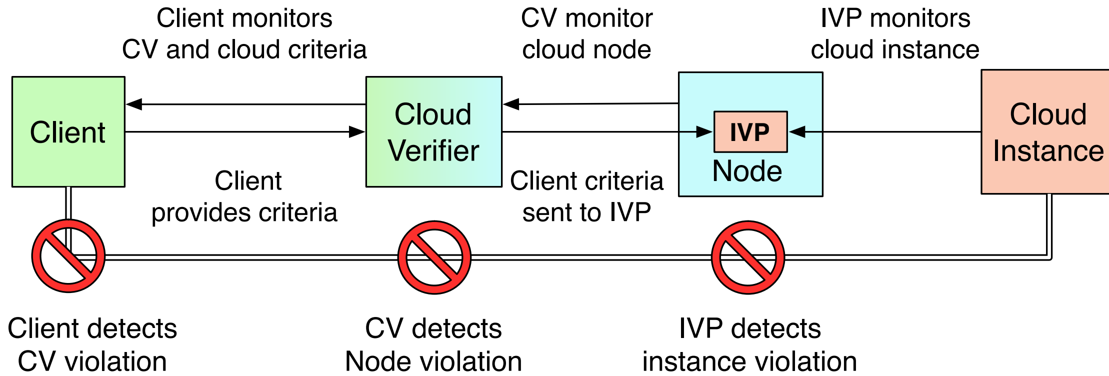

Fig. 2. Integrity Measurement Protocol

focusing runtime monitoring on security policy enforcement can approximate Clark-Wilson integrity for low monitoring overhead (less than 2% on the benchmarks we ran).

To verify the IVP platform's integrity, we use the Root of Trust for Installation (ROTI) approach to attest the filesystem of hosts that are modified infrequently [52]. At install time, a TPM-signed proof is generated that binds the installed filesystem to the installer and system image that produced it. Since host VMs modify few files at runtime, this proof will be useful across boot cycles. The ROTI method has been adapted for network installation, such as is done in the cloud [43]. In network installation, the phase of gathering the installer and system image may be under the control of an adversary, but the network-based ROTI mechanism produces of proof of exactly the inputs used to create the filesystem enabling remote clients to verify VM hosts upon which IVPs run.

5.2 Monitoring Cloud Integrity

Figure 2 shows the architecture of our integrity measurement mechanism for cloud computing. There are three key concepts. First, customers can provide *integrity criteria* dictating what the integrity requirements that must be satisfied in order to create a secure communication channel to the cloud instance. This enables the customer to dictate the terms of integrity to guide measurement, which is done using both traditional load-time and run-time (e.g., based on VM introspection) techniques. Second, the measurement framework use the IVP to track the integrity of the cloud instance and enforces customers' integrity criteria. We show elsewhere that it is possible to deploy IVPs on the cloud nodes to enforce a variety of criteria, including those based on CW-Lite [46]. Third, a *cloud verifier* provides the layer of indirection to enable the customers to use IVPs even though they may not know where their instances are deployed.

Figure 3 illustrates a simplified view of the changes to the basic OpenStack infrastructure to implement this approach. OpenStack is composed of various 'projects' that add additional services, but we will fo-

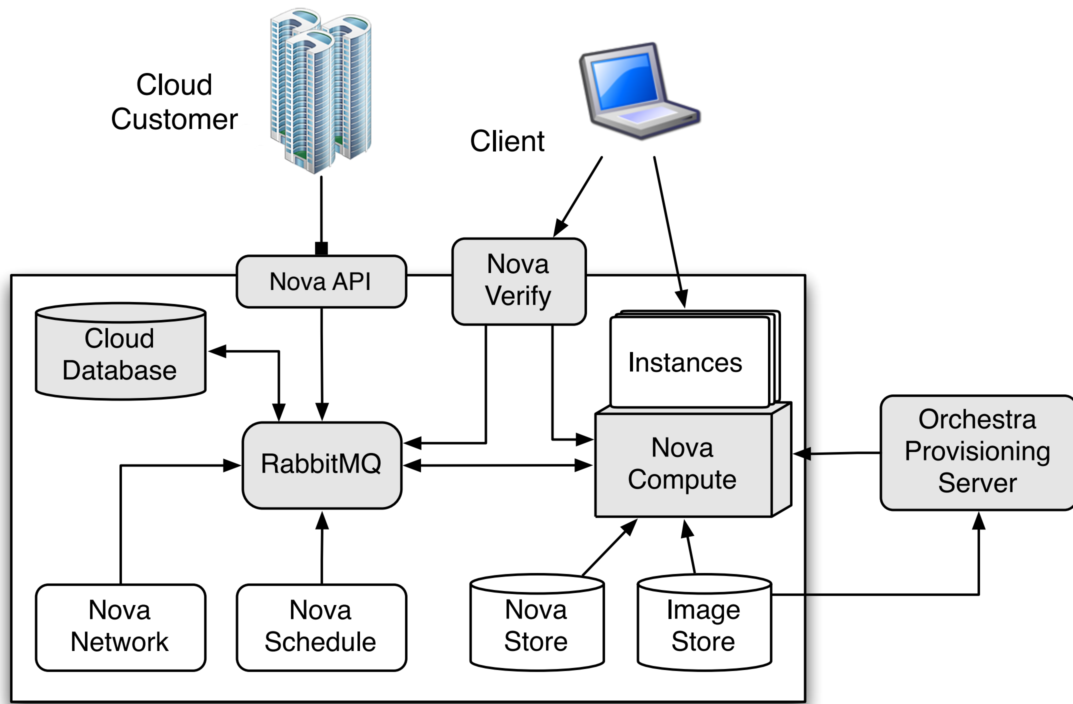

Fig. 3. Changes to OpenStack

cus on the `nova` project for our implementation because it provides the necessary components for hosting instances. The components are largely the same as those presented in the general IaaS architecture in Section 2, but some names have been replaced with their `nova` equivalent. The shaded components represent the changes we have made to implement the CV framework. First, the `nova-api` interface has been extended with a new commands to specify client criteria. Second, the cloud database has been updated to store node identity keys. Third, the `nova-compute` component (compute node) that hosts cloud instances was extended with our IVP implementation. We also modified the RabbitMQ message queue service to deny all messages except over an SSL channel authenticated by a CV signed certificate. This implements our requirement that only verified components can communicate through the RabbitMQ and thus participate in the cloud. Finally, the `nova-verify` service has been added as a standalone component to implement the CV. `nova-verify` is another public facing service in addition to the `nova-api` service. Overall, the additional code added to OpenStack was roughly 2,600 SLOC in Python. Additional code for implementing modules and measurement interfaces was under 1,000 SLOC of Python.

We evaluated this design on an OpenStack version 2011.3-nova-milestone cloud installed on three Dell M620 blades. These machines have two quad core Xeon processors with 64GB of RAM and two 1Gb network cards for the private and public network. The first blade serves as the compute node that hosts virtual machines. The second blade serves as the Cloud Verifier as well as other necessary controlling components of the cloud like scheduler, API server and network controller. The last blade is used to simulate clients of

Operation	Mean
CV Verification	1.09
TPM Quote	0.84
Communication	0.17
Others (read, write file etc.)	0.08
Node Join Protocol	1.68
TPM Quote	0.82
OpenSSL Node Key Generation	0.29
Node Certificate Generation	0.22
Communication	0.23
Others (read, write file etc.)	0.12
Client Criteria Registration	0.56
Openstack Processing	0.30
Instance Certificate Generation	0.22
Certificate Verification	0.04

Table 1. Protocol time delay breakdown. Times are in seconds and are averages of 30 runs.

the cloud and performs the benchmark programs. Each system runs Ubuntu-11.10 (x86_64) with a Linux 3.0 kernel for hosts and Ubuntu-11.10 (x86_64) with kernel 3.0.1 for virtual machines.

Table 1 shows the breakdown of three major protocols in our CV framework. These numbers are the average of 30 runs of the protocol. The first is the time due to verifying the `nova-verify` service, which the client must do before using the cloud. The second is the compute node's cloud join protocol. In both cases, the majority of the overhead comes from the TPM quote operation. Despite this, the delay is less than 2 seconds. For the node join protocol, this is a significantly shorter time than the boot process the node must go through and is only a one-time cost per boot. For the CV verification, we envision techniques like Asynchronous Attestation [27] can be used to to reduce this delay since it will be a major bottleneck for potentially thousands of clients connecting to the cloud. Finally, the client registration operation has a negligible overhead compared to the typical delay incurred by using the API server in general. It is worth noting that the total time will depend on the modules that must be checked for the client's criteria. This represents the minimum time only.

We benchmarked the G-WAN (G-WAN 3.3.28 64-bit) web server using ab (Apache Benchmark). We made 30 runs of the benchmark on the [1-1000] concurrency range. As shown Figure 4, the G-WAN web server running on the IVP framework performs at an average overhead of 3.1% compared to the G-WAN web server normally. We have found that this overhead is primarily due to VM introspection with `gdb` [46]. Since `gdb` uses the `ptrace` interface in the kernel to monitor processes for debug signals, every syscall incurs a small processing overhead by `gdb` to parse the signal and resume process execution. A possible solution for this would be to modify the `ptrace` interface to notify the `gdb` process only when debug signals are raised. This is future work.

6 Conclusions

In this paper, we describe a method and supporting tools for configuring cloud computations to protect their integrity proactively. Integrity protection is defined as an information flow problem, where any adversary access to an authorized operation has the potential to exploit a vulnerability. As security policies are too

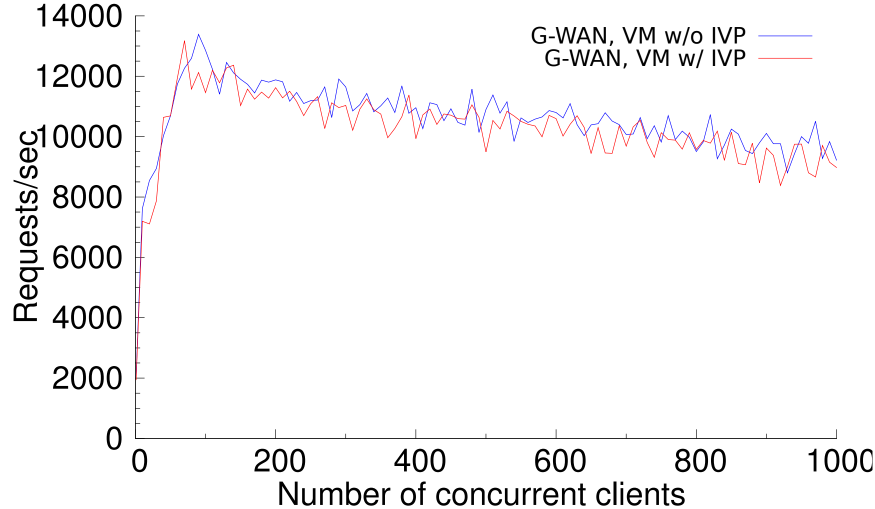

Fig. 4. G-WAN Server Performance

complex to evaluate manually, we describe a runtime analysis method to compute adversary accessibility to individual program entrypoints, a method that can compute the mediation necessary to resolve information flow errors system-wide given such adversary access, and defensive mechanisms to enforce that mediation approximating Clark-Wilson integrity. Using the resultant configuration, we define an integrity monitoring mechanism that uses trusted computing to enable cloud customers to ensure that their compute instances satisfy their integrity requirements. We demonstrate that this monitoring mechanism can be applied to monitor instances running in the OpenStack cloud platform for low overhead. In the future, we plan to extend our work in runtime vulnerability testing [58] to find and fix vulnerabilities in mediation.

References

1. Rackspace Cloud Servers, `http://www.rackspace.com/cloud/`
2. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity: Principles, implementations and applications. In: Proceedings of the 12th ACM Conference on Computer and Communications Security (November 2005)
3. Amazon Simple Storage Service(Amazon S3). `https://drive.google.com/`
4. Amazon EC2, `http://aws.amazon.com/ec2`
5. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity. In: Proc. 17th ACM Conference on Computer and Communications Security (2010), `http://doi.acm.org/10.1145/1866307.1866313`
6. Biba, K.J.: Integrity Considerations for Secure Computer Systems. Tech. Rep. MTR-3153, MITRE (April 1977)
7. Bleikertz, S., Groß, T., Schunter, M., Eriksson, K.: Automated information flow analysis of virtualized infrastructures. In: Proceedings of the 2011 European Symposium on Research in Computer Security. pp. 392–415 (2011)
8. Castro, M., Costa, M., Harris, T.L.: Securing software by enforcing data-flow integrity. In: OSDI '06: Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation. pp. 147–160 (2006)
9. Chen, H., Li, N., Mao, Z.: Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In: NDSS (2009)
10. Clark, D.D., Wilson, D.: A Comparison of Military and Commercial Security Policies. In: IEEE Symposium on Security and Privacy (1987)

11. Dhurjati, D., Kowshik, S., Adve, V.: Safecode: enforcing alias analysis for weakly typed languages. In: PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 144–157. ACM, New York, NY, USA (2006)
12. Garg, N., Vazirani, V.V., Yannakakais, M.: Multiway cuts in directed and node weighted graphs. In: in Proc. 21st ICALP, Lecture Notes in Computer Science 820. pp. 487–498. Springer-Verlag (1994)
13. Google App Engine. https://developers.google.com/appengine/
14. Google Docs. https://drive.google.com/
15. Howard, M., Pincus, J., Wing, J.: Measuring Relative Attack Surfaces. In: Proceedings of Workshop on Advanced Developments in Software and Systems Security (2003)
16. Jaeger, T., Sailer, R., Shankar, U.: PRIMA: Policy-Reduced Integrity Measurement Architecture. In: Proc. 11th ACM SACMAT (2006)
17. Jaeger, T., Sailer, R., Zhang, X.: Analyzing integrity protection in the SELinux example policy. In: USENIX Security Symposium (Aug 2003)
18. Jaeger, T., Schiffman, J.: Cloudy with a chance of security challenges and improvements. IEEE Security & Privacy (Jan/Feb 2010)
19. King, D., Jha, S., Jaeger, T., Jha, S., Seshia, S.A.: Towards Automated Security Mediation Placement. Tech. Rep. NAS-TR-0100-2008, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA (November 2008)
20. King, D., Jha, S., Muthukumaran, D., Jaeger, T., Jha, S., Seshia, S.A.: Automating security mediation placement. In: ESOP (2010)
21. Krohn, M.N., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles. pp. 321–334 (Oct 2007)
22. Levy, H.M.: Capability-Based Computer Systems. Butterworth-Heinemann (1984)
23. Manadhata, P., Tan, K., Maxion, R., Wing, J.M.: An Approach to Measuring A System's Attack Surface. Tech. Rep. CMU-CS-07-146, School of Computer Science, Carnegie Mellon University (2007)
24. Mannan, M., Kim, B.H., Ganjali, A., Lie, D.: Unicorn: Two-factor attestation for data security. In: 18th ACM Conference on Computer and Communications Security (2011)
25. McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An Execution Infrastructure for TCB Minimization. In: Proc. 3rd ACM SIGOPS/EuroSys (2008)
26. Microsoft Azure. http://www.windowsazure.com/en-us/
27. Moyer, T., Butler, K., Schiffman, J., McDaniel, P., Jaeger, T.: Scalable asynchronous web content attestation. In: Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC '09). ACSA (2009)
28. MSDN: Mandatory Integrity Control (Windows). http://msdn.microsoft.com/en-us/library/bb648648%28VS.85%29.aspx
29. Muthukumaran, D., Rueda, S., Talele, N., Vijayakumar, H., Teutsch, J., Jaeger, T., Edwards, N.: Transforming commodity security policies to enforce clark-wilson integrity. In: Proceedings of the 2012 Annual Computer Security Applications Conference (Dec 2012)
30. Myers, A.C., Liskov, B.: A decentralized model for information flow control. ACM Operating Systems Review 31(5), 129–142 (Oct 1997), http://www.cs.cornell.edu/andru/papers/iflow-sosp97/paper.html
31. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow. http://www.cs.cornell.edu/jif,July2001-2003
32. Necula, G.C., McPeak, S., Weimer, W.: CCured: Type-safe retrofitting of legacy code. In: Proceedings of the ACM Conference on the Principles of Programming Languages (January 2002)
33. NIST Definition of Cloud Computing. http://csrc.nist.gov/groups/SNS/cloud-computing/
34. Noel, S., Jajodia, S., O'Berry, B., Jacobs, M.: Efficient minimum-cost network hardening via exploit dependency graphs. In: ACSAC (2003)
35. Novell: AppArmor Linux Application Security. http://www.novell.com/linux/security/apparmor/, http://www.novell.com/linux/security/apparmor/
36. Security-enhanced linux, http://www.nsa.gov/selinux

37. Ou, X., Boyer, W.F., McQueen, M.A.: A scalable approach to attack graph generation. In: CCS (2006)
38. Pike, L.: Post-hoc separation policy analysis with graph algorithms. In: Workshop on Foundations of Computer Security (FCS'09). Affiliated with Logic in Computer Science (LICS) (August 2009)
39. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: USENIX Security Symposium (2004)
40. Salesforce. `http://www.salesforce.com/`
41. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE 63(9) (Sep 1975)
42. Sarna-Starosta, B., Stoller, S.D.: Policy analysis for security-enhanced linux. In: WITS (April 2004)
43. Schiffman, J., Moyer, T., Jaeger, T., McDaniel, P.: Network-based root of trust for installation. IEEE Security & Privacy (Jan/Feb 2011)
44. Schiffman, J., Moyer, T., Shal, C., Jaeger, T., McDaniel, P.: Justifying integrity using a Virtual Machine Verifier. In: Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC '09). ACSA (2009)
45. Schiffman, J., Sun, Y., Vijayakumar, H., Jaeger, T.: Cloud verifier: Verifiable auditing service for iaas clouds (Sep 2012)
46. Schiffman, J., Vijayakumar, H., Jaeger, T.: Verifying system integrity by proxy. In: 5th International Conference on Trust and Trustworthy Computing. pp. 179–201 (2012)
47. Shankar, U., Jaeger, T., Sailer, R.: Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In: Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium (February 2006)
48. Sheyner, O., Haines, J., Jha, S., Lippmann, R., Wing, J.M.: Automated generation and analysis of attack graphs. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy. p. 273. IEEE Computer Society, Washington, DC, USA (2002)
49. Shi, E., Perrig, A., van Doorn, L.: BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In: IEEE SP '05 (2005)
50. Shrivastava, S.: Satem: Trusted Service Code Execution across Transactions. In: SRDS '06 (2006)
51. Smith, S.W.: Outbound authentication for programmable secure coprocessors. In: In 7th European Symposium on Research in Computer Science (2002)
52. St. Clair, L., Schiffman, J., Jaeger, T., McDaniel, P.: Establishing and sustaining system integrity via root of trust installation. In: Proceedings of the 2007 Annual Computer Security Applications Conference. pp. 19–29 (Dec 2007)
53. Sun Microsystems: Trusted solaris operating environment - a technical overview, `http://www.sun.com`
54. TCG: Trusted Platform Module. `https://www.trustedcomputinggroup.org/specs/TPM/` (2005)
55. Vijayakumar, H., Schiffman, J., Jaeger, T.: A Rose by Any Other Name or an Insane Root? Adventures in Name Resolution. In: Proc. of 7th European Conference on Computer Network Defense (EC2ND). Gothenburg, Sweden (Sep 2011)
56. Vijayakumar, H., Schiffman, J., Jaeger, T.: Integrity walls: Finding attack surfaces from mandatory access control policies. In: 7th ACM Symposium on Information, Computer, and Communications Security (ASIACCS) (May 2012)
57. Vijayakumar, H., Schiffman, J., Jaeger, T.: Process Firewalls: Enforcing Safe Resource Access with Attack-Specific Invariants. Tech. Rep. NAS-TR-0153-2012, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA (Jan 2012)
58. Vijayakumar, H., Schiffman, J., Jaeger, T.: STING: Finding name resolution vulnerabilities in programs. In: 21st USENIX Security Symposium (2012)
59. Watson, R.N.M.: TrustedBSD: Adding trusted operating system features to FreeBSD. In: Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference. pp. 15–28 (2001)