CloudArmor : Protecting Cloud Instances by Validating Client Commands

Yuqiong Sun, Giuseppe Petracca, Hayawardh Vijayakumar, Joshua Schiffman and Trent Jaeger

6 August 2014
CloudArmor: Protecting Cloud Instances by Validating Client Commands

Yuqiong Sun
Department of Computer Science and Engineering
Pennsylvania State University
yus138@cse.psu.edu

Giuseppe Petracca
Department of Computer Science and Engineering
Pennsylvania State University
gxp18@cse.psu.edu

Hayawardh Vijayakumar
Samsung R&D
hvijay@cse.psu.edu

Joshua Schiffman
Advanced Micro Devices
josh.schiffman@amd.com

Trent Jaeger
Department of Computer Science and Engineering
Pennsylvania State University
tjaeger@cse.psu.edu

Abstract

IaaS clouds offer on-demand computing resources to clients to offload the burden of IT infrastructure management. However, in order to do so, cloud clients must place full trust in the cloud infrastructure, from various cloud services to the administrators managing the infrastructure. Such trust is often unjustified given the recent evidence of vulnerabilities found in cloud services and threats from accidental and intentional mismanagement by cloud administrators. In this paper, we introduce CloudArmor, a method that validates that client commands are executed as expected on cloud nodes, even if many cloud services are under the control of adversaries. Our insight is that cloud services act as a proxy that transforms commands to sequences of operations performed in cloud. Such transformations, however, do not transform argument values, so we can construct models that limit each command’s execution to legal sequences of system calls and predict their argument values. We implemented a prototype CloudArmor for the OpenStack cloud, converting the compute node to act as a proxy for client to validate how their commands are executed in cloud. Results show that CloudArmor can defend against a variety of attacks from cloud services, enabling a reduction of trust in cloud services by over 90% with no impact on the cloud function. Moreover, as CloudArmor only mediates cloud instance operations, it imposes less than 1.2% overhead for client instances. As a result, OpenStack clients can leverage CloudArmor to manage their instances safely without sacrificing performance or cloud function.

Keywords

cloud computing, security, trust

1. INTRODUCTION

One problem facing cloud computing is that several of the services deployed by cloud vendors to implement vital functionality are vulnerable to attacks. First, the Common Vulnerability Database [5] (CVE) lists more than 75 reported vulnerabilities in cloud services. For example, Amazon’s management interface [28] had a vulnerability that enabled attackers to issue arbitrary requests. Second, cloud insiders can mistakenly or maliciously misconfigure or abuse cloud services. DataLossDB [33] reports that 39% of data loss incidents in data centers are due to accidental misconfiguration of the computing environment.

Cloud service vulnerabilities impact clients because these cloud services are a built-in man-in-the-middle between clients and their instances. For example, when a client uploads a public key for logging in to her instance, a compromised service can hijack that command to insert a public key of its choice. Such an action would give the service (or the party controlling the service) unauthorized access to the client’s instance. For example, the attacker could perform a login, install a rootkit, and delete the public key, likely without being detected. Clients also depend on other cloud services, such as storage and database services, to provide data correctly for command arguments that the clients request. Currently, clients have no way to check whether the cloud services have implemented the clients’ commands on their instances correctly.

The standard approach to prevent man-in-the-middle attacks is to build a secure communication channel between the two endpoints, but a secure communication channel is necessary, but not sufficient, solution for eliminating attacks from compromised cloud services or malicious insiders. First, the client needs to determine the public key of a principal that is trusted to implement client commands on their instance to construct a secure communication channel for validating client commands. Researchers have developed methods to achieve this goal. Policy-Sealed Data [22] uses attribute-based encryption to enable clients to limit the release of their data only to cloud nodes whose platforms satisfy some criteria (e.g., geographical location, VMM version, etc.) that imply trustworthiness. Also, researchers have proposed Cloud Verifier services, which enable clients to validate that cloud nodes satisfy some criteria at runtime as well as load time [25]. We will lever-
key injection attack

age such methods to enable clients to create secure communication channels to send validation requirements to cloud nodes.

However, even if we can create a secure communication channel between the clients and a validated compute node that runs their instances, cloud services are allowed to transform client command messages into lower level operations (e.g., a sequence of system calls) and replace some command argument values. Thus, cloud nodes cannot simply verify that a command was transmitted securely, but they must verify that a transformation on that command was secure, which we call the command validation problem. It is this second problem that is the focus of this paper. While this problem is intractable in general, we find that it is practical to solve this problem in specific cases. For example, secure communication is a trivial form of this problem, where neither the message nor the arguments are allowed to be modified.

Our insight is that the transformations implemented for client commands can be predicted, so that it is possible to build a command validation method to enforce the predicted operations. Researchers have shown that it is possible to construct automata that represent the legal operation sequences for programs [35, 11, 10], but they have found that it is impractical to predict argument values in general [36]. However, cloud services are mainly designed to store and forward client inputs or cloud configuration values to helper processes that act on the instance. Thus, we find that cloud services can be constrained to the set of legal operation sequences and predictable argument values for those sequences given prior commands on that instance. Nonetheless, there remain a number of design challenges to overcome to validate client commands effectively. We must define a model for representing command validation policies, show how to track expected argument values when stored by the various cloud services, determine how to enforce those policies comprehensively for all cloud services and helper processes, and protect that validation method from tampering.

The outcome of this work is CloudArmor, a defense mechanism that protects client instances against compromised cloud services. CloudArmor is a reference monitor [1] that mediates all operations performed over client instances in cloud. It verifies that: (1) any operations performed over client instances are authorized by clients and (2) these operations are only performed in a constrained and client-approved manner. We implement CloudArmor as a kernel LSM module and evaluate it in the OpenStack [18] cloud. Further, we show that the rules for validating client commands can be obtained effectively using dynamic analysis and can be expressed accurately as finite state automata enforceable by CloudArmor.

Our work has the following novel contributions:

- We design CloudArmor, a mechanism for validating that client commands are transformed into system calls executed by cloud services as expected.
- We define an automata-based command validation policy model for express legal sequences of system calls and constraints on argument that predict their expected values and protect the cloud node from tampering.
- We implement and demonstrate CloudArmor for the widely-used OpenStack cloud. Our evaluation shows that CloudArmor: (1) protects cloud instances from compromised cloud services; (2) reduces the amount of cloud service code that must be trusted by over 90% ; and (3) causes less than 1.2% slowdown for client instances.

2. PROBLEM DEFINITION

In this section, we motivate our problem using an example attack, called a key injection attack, where a cloud service that is compromised or maliciously controlled by an insider may be used to gain unauthorized access to a client’s instance. We show that the fundamental cause of this problem is unnecessary trust in a variety of cloud services. We then define the precise problem that we aim to solve in this paper, ensuring validated command execution, demonstrating that current solutions are inadequate at present.

2.1 Key Injection Attack

Background. When client starts an instance in a cloud, she boots a system image (e.g., OS distribution) on a compute node. However, the system image is not sent along with the request, but rather, it is referred by an image name and retrieved from the cloud image service. These images are not assigned to a specific client. Any client can use these images to start up her instance. Clients will need to associate a set of credentials (e.g., public key for SSH connection) with the image when starting up instances, such that only those clients may perform a login to the instance later. To do this, the cloud dynamically injects SSH connection keys into the instance on behalf of the clients. The clients can then later perform a login using the corresponding private keys.

The detailed procedure for the OpenStack cloud is shown in Figure 1. First, the client issues the command nova keypair-add to cloud API service to create a key pair (i.e., mykey). While client keeps the private key, the public key will be uploaded to the cloud database. In the second step, client starts up a new instance using the command nova boot with mykey as the argument. The cloud API service queries the database to resolve mykey to the actual public key and sends it to the compute service. The compute service injects the public key into the instance image and boots up the instance. The client can then perform a login to her instance by establishing an SSH connection using her private key.

Attack. We now consider a malicious cloud API service. A malicious API service can modify the public key during either: (1) saving it to the database (Step 1) or (2) retrieving it from the database (Step 2). With a modified public key (e.g., owned by attacker) injected into client’s instance, attacker can thus log in to client’s instance as privileged user. An attacker could also launch a more subtle attack where he append his public key to the client’s, causing both to be loaded. An attacker could then install a rootkit in the instance and remove her public key before the client could detect the problem. Using some rootkit injection methods that rely on malicious data, this attack may even circumvent proposed trusted computing mechanisms to verify VM execution integrity [9, 2, 24] and be a challenge for VM introspection techniques to detect [19].

To launch the attack, attacker must be capable of modifying the public key injected into client’s instance. In the OpenStack cloud, unfortunately, there are many such opportunities. For example, the cloud database that stores the key and the message queue that delivers the key can both perform the modification, as long as attacker can compromise one of them. Worse even, OpenStack allows any cloud service to modify the cloud database at will. Consequently,
attacker only needs to compromise one of many cloud services in order to launch the key injection attack. More than 75 vulnerabilities [5] have been reported for OpenStack cloud services, and many of these enable the attacks against cloud instances.

**Attack Summary.** The key injection attack is merely one example of a number of possible attacks that may occur when cloud services are under an attacker’s control. In OpenStack, we found that client data are used in more than 40 client commands. Execution of these commands often involves more than one cloud service. In general, any protocol that utilizes untrusted services to store or convey data between two parties is a candidate for such attacks.

While man-in-the-middle attacks can be easily defeated, there are two challenges that make preventing such attack more difficult for client commands. First, clients may use multiple commands to achieve an effect, such as nova keypair-add and nova boot above, where the impact of one command may be utilized in the execution of another command. Second, cloud services transform the commands into lower level operations that implement such commands. For example, the OpenStack compute service executes system calls to inject the key pair into the instance. To ensure that commands are executed as expected, these two degrees of freedom must be constrained to comply with a trusted cloud system.

### 2.2 Problem Statement

The cloud service attack above can be formalized as follows. Clients submit commands from the set of possible commands \( C \). Cloud services implement each command \( c \in C \) as a sequence of operations \( O = (o_1, o_2, \ldots, o_n) \), where \( o_i \in O \), the set of all operations \( O \). We define a transformation function as a function, \( T: C \rightarrow \mathcal{P}^O \), from the set of commands \( C \) to the possible sequence of operations, which is the power set of all operations \( \mathcal{P}^O \), in the environment \( e \), which is the cloud environment consisting of the cloud services that request the operations. The problem is that \( T \) is implemented by many cloud services (e.g., compute service) together, some of which can be under control of attackers. Since the transformation is performed inside the cloud, the client cannot easily check that \( T \) was performed correctly by the cloud services. A compromised cloud service can easily take advantage of this ambiguity to launch the attacks described above.

**Validated Commands.** Suppose that the execution of a command \( c \) in environment \( e \) is the operation sequence \( O_{1,1} \), meaning that the \( T(c, e) = O_{1,1} \) is the transformation for \( c \) in the cloud environment \( e \) in which the operation sequence will be executed. A validated command for the command \( c \) in the cloud environment \( e \) is an operation sequence that composes with the transformation \( T(c, e) \). Ideally, compliance implies an exact match with the expected transformation. The main challenge here is to predict the transformation \( T(c, e) \) even before a particular command \( c \) is issued by the client.

**Current Alternatives.** Secure communication methods enable one party to validate a message sent by another party. Methods have been proposed to construct a secure communication channel between the cloud clients and the instance host [25]. However, this channel alone is insufficient to enforce validated commands in the cloud. An obvious problem is that secure communication methods require the message to be intact during transmission. In cloud, however, client commands need to be transformed into a sequence of operations performed by cloud services. Another approach to this problem is to leverage methods that validate that the data retrieved from untrusted storage is as expected. For example, client can validate the value of the key that is injected into its instance, therefore defending against key injection attack. However, resources checking alone are insufficient for expressing the transformations necessary for enforcing validated commands. Note that attackers can still tamper with the type and order of operations performed, despite that the resources themselves can be checked.

### 3. COMMAND VALIDATION METHOD

#### 3.1 Solution Overview

The goal of the design is to construct a method that solves the command validation problem for cloud. While the command validation problem is intractable in general, we find the transformation of client commands obey some key restrictions that enable the validation. Specifically, we find that cloud services perform very limited transformations on argument values. Either cloud services do not transform command argument values or they use command argument values to retrieve operation arguments. Thus, we only need to maintain an accurate mapping between name-value pairs used in retrieval to validate all argument values. That is, we can use program analysis to predict the possible, legal operation sequences for a command, as in host intrusion detection, and define constraints on argument values that enable comprehensive command validation for client commands on their instances even when transformed by cloud services. This insight enables us to construct a model for command validation policies expressed by automata analogous to the automata policies used for host intrusion detection systems [35, 11, 10], as described in Section 3.2. In the remainder of this section, we describe our command validation method for enforcing such policies, consisting of two parts shown in Figure 2: (1) a client-side validation method that collects constraints on argument values associated with client inputs and (2) a cloud node validation method that authorizes cloud service operations for each command using that command’s validation policy produced for the cloud environment and instantiated with the client’s argument value constraints.

**Assumptions.** In our design, we distinguish four types of principals: clients, cloud services, cloud nodes, and cloud vendors. We assume that clients are mutually-distrusting and each client does not trust cloud services for the reasons discussed above. On the other hand, we assume that clients trust the cloud vendors at the organization level. We assume that the cloud vendor may deploy methods at the organizational level to validate and justify trust in the cloud nodes that run client instances. Thus, we assume that the clients trust statements from the cloud vendor regarding the trust-worthiness of cloud nodes (i.e., public keys associated with cloud nodes) and the binding of cloud instances to those cloud nodes. Thus, the cloud vendor and validated cloud node systems (e.g., hypervisors and privileged VMs) form the trusted computing base of the design. Note, however, that we assume that cloud vendors and cloud nodes do not trust clients. Thus, while the goal of the work is validate that the cloud executes client commands as expected,

![Figure 2: Design overview.](Image)
the design must achieve this goal without allowing the clients to compromise the cloud nodes upon which their instances execute.

### 3.2 Command Validation Automata

In this section, we show how to extend the automata to express not only an authorized sequence of operations, but also restrict the possible values of operation arguments necessary to validate commands comprehensively using constraint templates. In theory, our method can apply to both pushdown and finite state automata, but we describe the method in terms of finite state automata for simplicity. A command validation policy is an automaton $M = (Q, \Sigma, \delta, q_0, F)$ where

- $Q$ is a finite set of states
- $\Sigma$ is an alphabet (finite and non-empty) for expressing operations
- $\delta : Q \times \Sigma \rightarrow Q$ is a transition function from one state to another as a result of an operation
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accepting states

An automaton models a cloud transformation function $T(c, e)$. Each client command maps to one automaton that describes how the command may be executed in the cloud environment $e$. Each automaton describes the legal operation sequences that may be executed to complete the command. The impact of the cloud environment is mainly in the form of constraints on operation arguments, as discussed below, which are compiled into the automata.

The core of the automaton is the transition function $\delta$. Transition function $\delta$ prescribes the set of allowed operations for every state $q_i$ in state space $Q$. An incoming operation will cause a transition from state $q_i$ to state $q_{i+1}$ if and only if it falls into the set of allowed transitions from $q_i$ to $q_{i+1}$. The set of allowed transitions are defined using the alphabet $\Sigma$ in the following way. An operation $o \in O$ is defined as a tuple, $o = (name, argspec_1, argspec_2, \ldots, argspec_n)$, where $name$ is the operation’s name and $argspec_1$ is an argument’s specification, which represents an operation’s arguments and constraints on the arguments’ allowed values as defined below. The number of argument specifications is the same as the number of arguments for the operation.

An argument specification is a tuple, $argspec = (argname, type, template_1, template_2, \ldots, template_n)$, where $argname$ is the name of the argument, $type$ is the argument’s data type, and $template_1$ is a constraint template that specifies the format of a constraint that may be applied to a particular argument value of that type. A constraint is essentially a predicate on the value of that argument, but templates are used because some predicates may not be known until the command $c$ and cloud environment $e$ are known. A constraint template is also a tuple, $template = (role, predicate)$, where $role$ defines the source’s role in the cloud (e.g., client) and $predicate$ defines the constraint test to be performed at runtime (e.g., $constrain_{t1} \equiv SHA - 1(value) \text{ or } constrain_{t1} > value$). A constraint template is a template for a predicate that can only be instantiated by a principal that satisfies the role. A principal that can prove that they belong to the role (e.g., hold a private key corresponding to a principal authorized for the role) may supply constraint values (i.e., $constrain_{t1}$) that will be used to evaluate the predicate on the argument’s value at runtime.

To enable command validation, we must define constraint templates for each argument in all operations to limit those arguments to predictable or safe values. We have found three types of constraint templates are necessary for cloud commands. First, several arguments use configuration parameters of the cloud, such as the instance image type (e.g., qcow2), which are defined when the cloud is configured. Constraint templates express the expected values for cloud configuration arguments (e.g., $value == constraint$ or $h(value) == constraint$, where expected value is $constraint$). Such constraint templates are built into the policy when constructed, see Section 4.1. Second, we predict values based on client inputs. Clients either input a value that should be used directly or a name that is used to retrieve the actual argument value. For example, when public key data is uploaded into a cloud instance, the data is retrieved from a record the client previously uploaded into the cloud database. Constraint templates express this linkage, using resource tuples defined in Section 3.3 such that constraints can be checked on values corresponding to uploaded resources (e.g., $h(valueof(resource)) == constraint$).

Third, we must also protect the compute node from unauthorized modification, so constraint templates must also be used to restrict argument values to those that are safe for the compute node. For example, when an operation creates a file on the compute node system (e.g., a temporary file), such files must not impact the integrity or secrecy of the compute node. Constraint templates express limitations on resource names chosen for servers to restrict them to safe names, such as ones that refer to resources that are not accessible to the node TCB (e.g., $resource \notin permissions(node)$, where $permissions(node)$ is the constraint).

When a command is run, the constraint templates for its automaton must be instantiated to create a command validation policy to be enforced for this specific command. Thus, enforcement involves two steps. First, we must enforce that each predicate constraint template is instantiated only by authorized parties. Second, in order for a transition to be taken, the proposed operation’s name must match one of the operations in $\delta$ for the current state and the values of all the proposed operation’s arguments must comply with the argspec instantiated predicate for that operation.

Figure 2 shows our method for validating client commands by enforcing command validation policies represented using the automata defined above. First, validation is initiated by clients when they submit commands on instances as detailed in Section 3.3. Second, the cloud-side validation method will ensure that each system call executed by cloud services either implements a valid transition in an instance’s command validation automaton or is not security-sensitive as detailed in Section 3.4. We further propose two refinements to the basic design in order to minimize the number of security-sensitive operations that must be validated: 1) We sandbox cloud services by only allowing them to run operations that act on resources of a client instance as described in Section 3.5 and 2) We carefully control how cloud services invoke any helper processes to perform operations on client instances as described in Section 3.6.

### 3.3 Instantiating Validation Automata

Whenever a client launches a command, the client-side validation method transmits the information necessary to instantiate an automaton for that command in the command validation policy. Instantiating an automaton requires that the client identify the command, the target instance for the command, and the constraint values necessary to instantiate the constraint templates. This information must be transmitted to the validation method on the cloud node securely, enabling the validation method to determine the client responsible for instantiating the command and that the instantiation data has not been maliciously modified.

First, when a client user submits a cloud instance command, the client-side validation method produces a command instantiation message, which must include: (1) the command name; (2) the target instance; and (3) the client’s argument value constraints for instantiating the command validation policy for that command’s exe-
The command name is necessary to identify the particular command validation policy (i.e., choose the automaton). The target instance is necessary to associate the command’s validation with the proper instance. The client’s argument value constraints are necessary to instantiate the automaton by setting the constraint values in the constraint templates of the automaton.

The client faces a two challenges in producing command instantiation messages. First, the client must determine the identity of the instance in order to enable validation of operations on the correct instance. However, this is a chicken-and-egg problem, as the instance is created as the result of a client command. To address this problem, the validation method must return the identity of cloud resources created as a result of the execution of valid commands to the client. To do this, the validation method returns resource tuples, resource = (value, type, server, name), for each successful command to associate resources identities with their expected values. For example, the creation of a new instance would create a tuple, resource = (vm_handler, instance, host_id, inst_name), which associates the VM handler seen by the hypervisor to an object of an instance type where this instance is identified by inst_name and the server that generates the resource is identified by host_id. Resource tuples are produced in three ways. First, they can be produced as a result of clients storing resources in cloud (e.g., keys). In this case, correct values of resources are known to clients since clients created them. Second, resource tuples can be produced by verifying resources created by cloud administrators (e.g., flavor1). Although in this case, the resources are created by an untrusted party (i.e., cloud administrators), clients can produce the tuple by verifying if the resources are acceptable (e.g., only create the resource tuple if the flavor is acceptable). Finally, resource tuples can be produced by the validation method as a result of the execution of previous commands on client instances (e.g., creating instances). Resources tuples represent client specific constraints over cloud resources. Once produced, they are stored locally at client side, preventing cloud services from tampering with them.

Second, the client must pick constraint values for operation arguments to instantiate the constraint templates. The idea is that the client will use the command inputs or resource tuples obtained from prior commands to predict the client-defined argument values for operations. These will be collected into a set of variable-value pairs which are called a constraint instantiation, where variable is the name of a constraint variable in the command templates and value is the constraint value for this command. For example, when uploading public keys to the cloud, the command nova keypair-add uploads the public key to the cloud database, creating the resource tuple (key_value, key, client, key_name). Later, when that public key is to be uploaded into a specific instance, the client-side validation method retrieves the resource tuple for any resource names specified in the command and instantiates the constraint value in the constraint template for that resource name.

Once the client has collected the instance and constraint instantiations, the client constructs a command instantiation message and must convey that message to the validation method on the cloud node running the instance for enforcement. We leverage existing methods for distributing the public key of a cloud node running a client’s instance to a client. Such methods leverage a root of trust in the cloud, called the cloud verifier [25]. The cloud verifier serves two roles typical of a root of trust. First, the cloud verifier validates the integrity of any compute node prior to allowing it to join the cloud. Second, the cloud verifier is the root certificate authority for the cloud, enabling the construction of a chain of trust to the validation method of each compute node and to the client’s instance. Once the client has the public key for the verification method on the node running their instance, standard techniques (e.g., SSL/TLS) are employed to construct a secure communication channel.

3.4 Validating Cloud Service Operations

Operations that may impact instances by loading or modifying the instance execution environment and by extracting data from running instances are fundamentally performed as system calls on the cloud node running the instance. Thus, we aim to extend the operating system on the cloud node, which executes these system calls with a reference monitor to enforce our command verification policies over these system calls. Note that as described in the trust model, we depend on the cloud node’s trusted computing base, including the operating system on the cloud node, to not be under adversary control when the cloud node is launched (e.g., validated using trusted computing methods [23]).

Researchers have a wealth of experience developing reference monitors for system call authorization, several extensive reference monitors are now available in commercial systems [38, 34]. We leverage such available mechanisms, in particular, the Linux Security Modules (LSM) framework in the Linux kernel [38] to provide reference monitor guarantees. Using LSM, our validation method is invoked on all system calls that may ever be security-sensitive. Our challenges then are to determine which system calls may impact a client instance, find the command associated with those system calls, and validate that the system call complies with that command using instantiated command validation policy automaton for that command and instance.

To determine which system calls may impact a cloud instance, we begin by identifying which resources belong to cloud instances2. In order to do so, we take a conservative approach that utilizes the available mandatory access control (MAC) enforcement. In general, the idea is that if the cloud service executes a system call that causes an information flow to or from a system resource to which the instance has access, then that system call is a security-sensitive. To identify instances access to system resources, we leverage the sVirt project [29], which associates an instance-specific label to all resources associated with each client instance. Using this approach, we can detect whether a particular system call invoked by a cloud service accesses cloud instance resources and identify the instance associated with those resources.

Once the instance has been identified, the system call invocation can be validated using the instantiated command validation policy for that instance. Each system call must correspond to a legal transition from the current state of the policy’s automaton. Recall from Section 3.2 that this means that the operation name must match the system call name and each of the argument specifications must be satisfied. The constraint values supplied by the client at instantiation time and the other defined at compile time (e.g., for the cloud configuration) are enforced to validate the operation.

Should a system call create a new resource, the validation method on the cloud will create a new resource tuple (Section 3.3) in addition to validating that the resource satisfies constraints. Such resource tuples are returned to the client validation method to instantiate command validation policies for future commands.

3.5 Sandboxing Cloud Services

As prescribed by Anderson in his definition of the reference

---

1Flavor is the notion of instance type in OpenStack, used to specify CPU amount, memory size and etc..

2We note that system calls may impact client instances indirectly (e.g., by accessing privileged resources or other processes). We address these cases in subsequent sections.
monitor concept [1], we need to protect the validation method from tampering. One major threat to the validation method is that the cloud services may use operations to compromise privileged processes on the compute node, enabling privilege escalation to circumvent or disable the validation method. We will refer to these processes as root processes, as root is historically the privileged principal on UNIX systems. The cloud services under control of an attacker may use permissions to IPC or invoke a root process, plant a file used by a root process (e.g., file squatting or untrusted search path), or various other means to launch a local exploit that may compromise the root process, giving the access to instance resources and the ability to disable the validation method entirely. We must prevent such attacks.

We envision that there are two plausible solutions to mediating the cloud services’ access to root processes. First, we could identify operations that may impact root processes and include them in the command validation automaton. Such operations would utilize permissions accessible to root processes, which in general are all permissions. Mandatory access control (MAC) (e.g., SELinux [17]) can limit root processes to least privilege permissions. Using such MAC policies, the privileged resources we must protect are those that may be written by cloud services and read by root processes, because cloud services can access such resources and they may impact the integrity of the root processes. Accesses to operations that access such integrity-critical resources would need to be included in the command validation automaton.

Alternatively, we could run cloud services in a separate environment in which they would be prohibited from accessing these integrity-critical resources (i.e., sandbox the cloud services). A sandboxed cloud service is given its own copy of resources, which may access freely, but the cloud service may not access the real node resources, protecting them from malicious modification and protecting node processes from compromise. Further, sandboxes may grant limited privileges to real node resources to the sandboxed process to permit some limited actions to be performed.

As validating process access to integrity-critical resources is a complex problem [36], we instead choose to sandbox the cloud services and give them access only to instance resources on the cloud node. Thus, the sandbox includes a system environment sufficient to run the cloud service (e.g., a virtual machine or container). However, the sandboxed environment detects whether a system call targets an instance resource, as described above, and forwards only those system calls to the cloud node. Other system calls are run within the sandbox, protecting other integrity-critical resources. Researchers have constructed mechanisms for system call forwarding for virtualized environments (e.g., Proxos [31]) that we leverage for our sandbox. While such forwarding has non-trivial overhead, client commands are infrequent and not run on the critical path of instance processing. For system calls that are forwarded to the cloud node, we will detect attacks on other client’s instances by enforcing the command validation policies as described in the previous section.

Finally, we must ensure that new resources created by cloud services do not compromise the integrity or secrecy of the cloud node itself. Naturally, such actions could enable the cloud services to compromise the cloud node. As described in Section 3.2, we apply constraint template to arguments that contain the name of a new resource to prevent access to cloud node resources. Using LSM, we can authorize access to individual inodes to prevent various attacks on name resolution (e.g., TOCTTOU).

3.6 Validating Helper Processes

As a result, we only need to validate operations from cloud services that act on instance resources. However, cloud services may use helper processes in the cloud node system to modify instance resources. For example, compute service uses iptables tools to manage the firewall rules for instances. As another example, compute service may use Libvirt or Xen tools to manage (e.g., start and stop) client instances. The challenge is to invoke helper processes in a way that ensures the operations on instance resources will be performed as expected and that integrity-critical resources are protected from those helpers.

While in theory we could include helper process system calls in the command validation automaton, in practice it is often impractical to validate all the argument values used in these system calls [36]. For example, compute service invokes iptables-restore tool to update the firewall policy on the compute node. The input to the tool is the firewall policy specified by the client, which can be validated. However, the iptables-restore tool executes several hundreds of system calls, many of which are dynamic and dependent on runtime system. Failure to validate these system calls can potentially lead to mimicry attacks [36].

The foundation of our solution is to validate the cloud service operations that launch and communicate with helper processes. Since the cloud services are forwarding client inputs to the helper processes, they may be easier to validate the argument values at the cloud service. However, to ensure that the helper processes will perform as expected we need to validate that the helpers run approved code only (including libraries) and that all interactions with cloud services are measured. To do the former, we use typical code validation techniques of validating the hash value at all code load operation in the kernel for the helper processes [21]. To do the latter, all system calls that operate on helper processes resources must be mediated. Solving this problem is the same as checking for access to instance resources, which we detailed above.

We also need to protect the operating system that runs the helper processes from the cloud services. Since we ensure that the argument values correspond to client inputs, the way in which the helpers are launched is trusted by the client, but is not necessarily trusted by the cloud node. There are two choices: either we run the helpers in a separate sandbox or we run client-specific nodes, as proposed in the Self-Service Cloud (SSC) architecture [4]. The SSC architecture creates service domains in each compute node for each client that the cloud infrastructure is isolated from. We expect to apply the SSC architecture, which gives the clients freedom to manage their cloud instances without privileges that may tamper the cloud node. However, this implementation is future work.

4. IMPLEMENTATION

Figure 3 shows an overview of our implementation for command
validation on the OpenStack Grizzly release. At a high level, we implement mechanisms for automating the generation of command validation policy automata (training) in OpenStack clouds and enforce such automata at runtime (production cloud).

4.1 Training

In our implementation, we use dynamic analysis to generate command validation policies targeting OpenStack clouds. While researchers have shown that it is possible to generate automata statically from program code [35, 11, 10], a problem we face is that multiple cloud services interact to transform client commands to operations over client’s instance. Dynamic analysis is practical to execute, but may be incomplete, leading to false positives. We evaluate false positives for OpenStack given our approach in Section 5.3. Exploring alternative methods to determine this statically or by combining static and dynamic analyses is future work.

To produce command validation policies automatically, we must setup a training cloud, log the commands execution over the training cloud, and convert the logged traces into command validation policy automata that include constraint templates. The training cloud represents a trustworthy version of the OpenStack cloud system. While ideally the training cloud would run the exact software as production cloud, we only restrict that cloud services and configurations are the same. If another program impact the legal behavior, we could make training dependent on this program as well.

Over the training cloud, we use the Trace Logger to collect traces of system calls that are issued for executing client commands. In OpenStack, we find that only system calls issued by the compute service need to be logged. This is because OpenStack cloud, and many other cloud as well, operates over client instances via agents (e.g., compute service) running on the compute node. The Trace Logger logs all the compute service’s security-sensitive system calls along with their arguments. We have enumerated all possible variants (i.e., different options) of every client command to produce a comprehensive set of traces, and we describe the effectiveness of this method in Section 5.

Given a trace set associated with each client command, the goal is to compute the correct representation of transformations (i.e., from client command to system calls) embodied by these trace sets suitable for runtime validation. We built a Trace Analyzer that converts logs, outputted by the Trace Logger, to nondeterministic finite automaton (NFA). One automaton for each client command. The Trace Analyzer leverages a tool, called Synoptic, that builds NFAs from traces [3] by merging linear graphs in a manner that satisfies three types of temporal invariants (always followed by, never followed by, and always precedes). Using these invariants, the Trace Analyzer eliminates impossible paths from the NFA, avoiding eventual false negatives.

Looping is a particular aspect that needs to be addressed. In the trace sets, some system calls may be issued multiple times in succession. For example, compute service issues `stat` system call to check the status of a downloading instance image. The resultant FSAs allow repeated system calls to be rerun in loops of arbitrary length. However, we note that such denial of service attacks can be easily detected by cloud vendors.

The FSAs generated by the Trace Analyzer encodes constraint templates, as defined in Section 3.2. When merging traces of a client command, the Trace Analyzer identifies arguments whose values remain constant no matter which client command options are specified. The values of these arguments are built into the automata as fixed constraint templates. For OpenStack, we found that, an average of 86% of argument values are fixed. A simplified FSA example is shown in Figure 4. In this example, arguments such as `qemu --img -f` are fixed for the training cloud. Other arguments are dynamic arguments that either must be instantiated from client input or are resource names whose safety must be enforced. Identifying client input is straightforward, as we test for dependency between command arguments and values. Where client input are resource names, resource constraints are instantiated. The remaining arguments we saw in OpenStack are resource names chosen by the cloud services, which we restrict to safe values to protect the compute node as described in Section 3.5.

4.2 Production Cloud

The cloud-side command validation method we built is called CloudArmor. It validates the security-sensitive system calls against the command validation automata. CloudArmor is constructed as a Linux Security Module [26] (LSM) to intercept system calls forwarded by the sandbox mechanism and validate them against the automata. Upon violation, CloudArmor simply aborts the execution of the system call and raises an alarm. In future, we plan to explore more flexible remediation mechanism such as rollback.

The client-side validation method implements instantiation of command validation policies and collects resource tuples from prior commands, as described in Section 3.3. We modified nova CLI tools such that command validation policies are instantiated when client commands are entered. We use the open-source Cloud Verifier code integrated with OpenStack [25] to establish a secure channel between clients and compute nodes in order to transfer the instantiated command validation policies as well as resource tuples created at compute nodes.

Finally, we leveraged the Linux container [16] to implement the sandbox mechanism. The linux container allows a sandboxed process to run on the same kernel as the host while creating a different namespace (e.g., file, IPC, network) for the sandboxed process. We modified the Linux kernel such that different namespaces are presented to the compute service upon execution of different system calls. System calls that access files labeled under an instance or access `libvirt` are redirected to the host (i.e., compute node) namespace. System calls that invoke a new helper processes (e.g., `clone`, `execve`) are a bit more complex. After `clone` system call creates a new helper process, subsequent system calls of the helper process until `execve` will continue running in the compute service’s namespace. We redirect `execve` system calls to the host namespace and before the `execve` commits, we move the entire helper process out of the compute service namespace. Subsequent system calls from the helper process will be executed in the host namespace thereafter. Note that in this case, the only system call that must be validated against the automaton is the `execve`. Since the helper processes and the compute service now run in different namespaces, the compute service cannot tamper with the way that the helper processes execute.

5. EVALUATION
5.1 Automaton Complexity and Precision

We demonstrate the complexity of our model by showing the number of states and transitions of each automaton (Column Auto. Complex. in Table 1). Even for the most sophisticated command - migrate, the model contains only 57 states, 71 transitions. The sandbox mechanism allows to reduce considerably the automata complexity. Column # of Syscalls in Table 1 shows the average number of system calls issued by the compute service and the number of system calls that are redirected to the host namespace. As shown in the table, the majority of system calls are contained within the compute service namespace and therefore they do not need to be included in the automata.

We measure the precision of automata using two metrics. First, we use the Wagner’s average branching factor metric [35]. This metric measures the average number of possible transitions after each state in the automaton. Average branching factor is a useful approximation for the ability of a model to defend against mimicry attacks [36]. Smaller numbers are favorable as attackers are more constrained and have less freedom. Column Avg. Bran. Fac. in Table 1 shows the average branching factor for different client commands. Even for migrate, the average branching factor is close to one. Second, we measure the percentage of predictable arguments in security-sensitive system calls. This includes both the cloud constraints from the training and client constraints applied at runtime. More predictable argument values means more precise automata. As shown in column % of Args Predicted, an average of 90% of the arguments in security-sensitive system calls can be predicted. The reason behind this is that most of the system calls seen by the CloudArmor are `execve`. The arguments to the `execve` are predictable due to the way the helper processes are invoked. They are only dependent on the client command and the cloud configuration.

The precision of the automata are closely related to security. The more tightly we constrain security-sensitive system calls, the less freedom is provided to attackers to compromise cloud services. In an ideal case, a client command would be transformed into a unique sequence of system calls with determined sets of arguments. If this is the case, most compromised cloud services are given zero freedom, as the execution of client command is entirely decided. As we show in this section, the models built by CloudArmor framework are very close to the ideal case. They have an average branching factor close to one and an average of 90% arguments predictable in the automata. The other 10% of arguments whose values cannot be predicted are temporary files and file descriptors, dynamically created by the compute service at runtime. As discussed in Section 3.5, we place constraints over these arguments such that they can cause no harm to the compute node system.

5.2 TCB Reduction

An important effect of our approach is that it eliminates the need to trust several cloud services. This results in a TCB orders of magnitude smaller. Table 5.2 shows a TCB comparison between a vanilla OpenStack cloud and a CloudArmor OpenStack cloud. Results show that CloudArmor can effectively remove most of cloud services from the cloud TCB, reducing the cloud TCB down by 90%.

We measured the false alarm rate of our framework in two ways. First, we measured the false alarm rate using a hand built training set and test cases from OpenStack Tempest test suite [32]. The false alarm rate we obtained is close to zero. However, we note that OpenStack Tempest is very limited and contains only few test commands that we can utilize. Hence the false alarm rate may not apply to real world clouds. The practical difficulty is that other available OpenStack test suites are also very limited. In future, we plan to test our framework more comprehensively using real world workload (e.g., deploy it to a class where students can use the cloud as normal clients). Second, we are interested in measuring the relation between false alarm rate and the comprehensiveness of the training. The relation could indicate how much effort is required in order to deploy our framework.

Figure 5 shows the relation between the number of training runs performed (X-axis) and the detected false alarm rate (Y-axis). We started the training from a small number of command variations picked from the hand built training set. To evaluate the false alarm rate, we use the OpenStack Tempest test suite as test cases. We then
increase the number of training runs and record the corresponding false alarm rate change. Results show that for simple commands such as delete, only 10 training runs are needed in order to achieve a high coverage. For complex commands such as boot, less than 20 training runs are required. The result is in line with what we expected. Cloud is supposed to provide stable services to its clients. Given a fixed cloud configuration, the operations performed should exhibit minimal uncertainties and in an ideal case, be only affected by client commands. Consequently, few training runs that enumerate possible command variations could achieve high coverage. What this means is that CloudArmor framework requires only little training efforts and can be quickly deployed into real world clouds.

5.4 Performance

We evaluate the performance of our framework by measuring the impact on client’s instances performance. We are interested in the slowdown of client instances as a result of integrating CloudArmor. Our experiment setup involves 1) a vanilla OpenStack compute node and 2) a CloudArmor compute node. Both compute nodes run OpenStack Grizzly with KVM. Client instances are configured with 2 VCPUs, 4 GB memory and 10 GB virtual disk. Each compute node only hosts one instance during the experiment. Following benchmarks are used to measure the performance of instance: 1) Kernel Build (KBuild) for CPU intensive workloads, where we measured the compilation time of Linux kernel 3.0.1; 2) Apache for network I/O intensive workloads, where we measured the throughput (request/sec) of Apache web server 2.2.20 using Apache Benchmark (ab) to make 100,000 requests at a concurrency level of 1000; 3) dbench for disk I/O intensive workloads, where we measured filesystem I/O throughput (MB/sec) with 20 concurrent clients. For a microbenchmark, we used lmbench. The results are shown in Table 3. For various benchmarks, the maximum slowdown for an instance is less than 1.2%. The overhead is relatively small because CloudArmor is not on the critical path of instances: CloudArmor only mediate system calls issued as a result of client commands and client commands are infrequent.

6. RELATED WORK

We now present a summary of work related to protecting cloud instances, why they are insufficient for meeting our goals, and how the CloudArmor overcomes their shortcomings.

Application Integrity. A line of research has been to guarantee the integrity of cloud resources. Several projects aim to verify the integrity of workflow processing in specialized cloud environments like IBM’s System S stream processing cloud [7]. These approaches use application level attestation protocols to enable detection of where illegal modification have occurred in the data flow. Other techniques like PeerReview [13], Accountable VMs [12], and Trinc [15] leverage hardware-based attestation to verify adherence to known distributed application protocols and detect Byzantine faults. However, these approaches focus on application protocol verification and cannot discover violations that are outside of the protocol’s scope such as key injection attack.

Storage Security. Research by Wei et al. aims to protect the instance disk image in the cloud by alerting the user to unauthorized modifications [37]. The Silverline project [20] uses a set of analysis tools to identify functionally encryptable data, data that can be processed as cipher text, to improve data confidentiality in an untrusted cloud platform. These data security techniques can certainly complement any cloud-hosted application, but they provide little guarantees for the operations performed in cloud.

Virtual Machine Security. Research in hardening hypervisors from both co-resident VMs and insiders has become popular. Approaches like NoHype [30] minimize the VMM’s attack surface by eliminating all but a small resource management kernel. Other techniques like formally assured L4 microkernels [14], CloudVisor [39] and privileged domain separation [31] again limit the impact a compromised hypervisor or privileged domain can have on hosted instances. Finally, low-level monitors like DeepSafe [6] use the protected System Management Mode memory region to monitor and enforce hypervisor integrity at runtime. These works are complementary to our work. By combining these techniques with CloudArmor framework, we can better protect client VM against attacks originated from both cloud services and the VM host.

Host based IDS. CloudArmor framework shares many ideas with host based intrusion detection systems. Host based IDS builds a behavior model of program via static [35, 11, 10] or dynamic [8, 27] analysis and then checks runtime program behavior against the model. CloudArmor follows the similar idea. However, unlike these approaches, CloudArmor does not build a model upon full set of system calls. Instead, it only uses a small set of system calls that may potentially harm client VM. This greatly reduces the model complexity in CloudArmor.

7. CONCLUSION

In this paper, we have presented CloudArmor, a framework that enables clients to enforce how their commands are executed in cloud. CloudArmor framework constructs a behavior model of cloud through performing dynamic analysis over cloud, taking into consideration of specific cloud configurations. Then by enforcing the behavior model in cloud, CloudArmor ensures that the execution of client command is as expected. We demonstrate that CloudArmor framework can defend against a variety of attacks from compromised cloud services without affecting cloud functionality and hurting performance of client instances.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1117692. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES
