CloudArmor: Protecting Cloud Commands from Compromised Cloud Services

Yuqiong Sun, Giuseppe Petracca, Trent Jaeger
Penn State University
Email: yus138, gxp18, tjaeger@cse.psu.edu

Hayawardh Vijayakumar
Samsung R&D
Email: h.vijaykuma@samsung.com

Joshua Schiffman
Advanced Micro Devices
Email: Josh.Schiffman@amd.com

***Abstract*—Infrastructure-as-a-Service (IaaS) clouds can be viewed as distributed systems of cloud services that are entrusted to execute users' cloud commands to provision and manage clouds computing resources (e.g., VM). However, recent vulnerabilities found in cloud services show that this trust is often misplaced. By exploiting a vulnerability in a cloud service, an adversary can hijack or forge commands to modify user VMs, exfiltrate sensitive information, and even modify other service hosts. This paper introduces CloudArmor, a system that detects and blocks the tampering of user commands without the need for modifications to cloud services. Our insight is that we can construct state machine models to limit the system call sequences executed by cloud services. By applying constraints over system call arguments, we can restrict the way user commands are executed, blocking unauthorized operations from compromised cloud services. We implemented a prototype CloudArmor system for OpenStack, a widely adopted open source cloud platform. Results show that CloudArmor can greatly limit attack options available for adversaries while imposing less than 1% overhead for user VMs. As a result, cloud users can leverage CloudArmor to execute user commands safely even in presence of compromised cloud services.**

***Keywords*-cloud computing; security; state machines;**

I. Introduction

Cloud computing has revolutionized they way we consume computing resources. Instead of maintaining a locally administered data center, cloud users obtain resources on demand from a public cloud utility offered by several cloud vendors [1], [2]. Cloud vendors enable their users to provision and manage cloud resources through a command API (e.g., the REST API in OpenStack), instances of which we will call *user commands*. Such commands enable users to remotely obtain virtual machines (VMs) on cloud nodes, select execution images to launch, assign storage volumes to the VMs, and configure security of the VMs (e.g., determine who can login, configure firewall rules, etc.).

Cloud users envision that the commands that they submit to a cloud vendor will be executed as they specified, but vulnerabilities in cloud services put such command executions at risk. Current cloud architectures deploy a variety of *cloud services*, which are responsible for managing different types of resources, such as VMs, images, storage volumes, etc. Each user command may be processed by a variety of services. For example, the command to upload a user's public key into a user VM on the OpenStack cloud (e.g., to enable the user to login to her VM via SSH) involves an API service (to authenticate the user), a database service (to retrieve the public key file), and a compute service running of the cloud node hosting the user's VM. If a vulnerability in any of these services enables an adversary to modify command processing, then an adversary may be able to subvert the command (e.g., load an adversary-chosen public key). To date, well over 100 vulnerabilities have been reported in OpenStack cloud services [3], so cloud vendors would benefit from a method that detects attacks on such command processing.

Currently, cloud architectures lack defenses to detect attacks on user commands. Cloud architectures assume that the cloud services are all trusted components in the cloud system. However, cloud services are often complex web applications running on conventional systems, leading to the large number of vulnerabilities reported. Researchers have identified that current cloud architectures suffer from similar design flaws to conventional operating systems circa 2000, where large amounts of complex, fully trusted code enabled a single vulnerability to bring down the entire system [4]. However, current research focuses primarily on attacks from one VM to another on the same cloud node [5], [6], where defenses proposed to protect user command processing only address image anomalies and data release [7], [8], not malice by compromised cloud services.

Cloud users want to believe that their commands are run as expected, and cloud vendors would like to detect attacks against their clouds without significant modifications to their cloud architectures. To achieve these goals given the current cloud architecture, we leverage the following insight. Cloud services execute user commands by: (1) collecting the respective data referenced in the commands and (2) using a series of programs on the VM host to apply that data to the user VMs. Thus, we propose a two-stage defense. First, we check that the data applied by cloud services in executing a user command corresponds to the data referenced in the original user command. Second, we check that the sequence of programs chosen to process that data is expected for that command. With such restrictions, even a fully compromised cloud service is quite restricted in the attacks it could launch against user commands without detection.

In this paper, we explore whether the cloud node's host operating system can be modified to enforce the two-stage defense proposed above. To model the possible sequences

of programs executed for each command, we propose using a (nondeterministic) finite state machine. The idea is to limit each command's execution to the possible sequences of commands implied by the command's state machine. To restrict the possible data values that may be chosen by services, we annotate the state machine transitions with constraints that are instantiated from the user command. These constraints specify the possible values for the user-supplied arguments used by cloud services. The key experiment is to determine whether it is possible to generate such state machines in a manner that restricts the attacks significantly without producing false positives.

In addition, once we assume that cloud services are untrusted and/or performing operations as dictated by remote users, we want to add defenses on the cloud node to protect the cloud node's VM host from the user commands. In current cloud architectures, the cloud services that run on cloud nodes may perform privileged operations on the host VM. However, if we do not trust cloud services and users can get cloud services to execute privileged programs, then users can perhaps use flaws in cloud services to attack the VM host. Thus, we also isolate the compute node's cloud services from the VM host by running those services in a VM as well. We only forward requests to the VM host when necessary to implement a command and these requests are limited by the state machine above.

Our work has the following contributions:

- We define a method for detecting attacks on user command execution using a state machine model. Our model features a branching factor of 1.22 or less with causing false positives in our experiments.
- We design the CloudArmor System, which restricts system calls issued by cloud services using this state machine model, preventing compromised cloud services from performing unauthorized operations.
- We deploy the CloudArmor system in the widely-used OpenStack cloud. Our evaluation shows that CloudArmor system can greatly limit attack options available for adversaries, while causing less than 1% performance overhead to user VMs.

II. Background and Problems

A. OpenStack Background

In this experiment, we will examine defenses for the OpenStack, which is an open-source, cloud software stack for building public or private IaaS clouds [9].

Figure 1 shows the OpenStack architecture evaluating a user command (see below). OpenStack embraces a modular architecture, where each *module* in OpenStack is an independent project for providing a suite of services to cloud users to flexibly manage their VMs and the cloud resources that their VMs depend on. For example, in OpenStack `Nova` module manages VMs, `Neutron` module manages networks, `Glance` module manages images and so on.

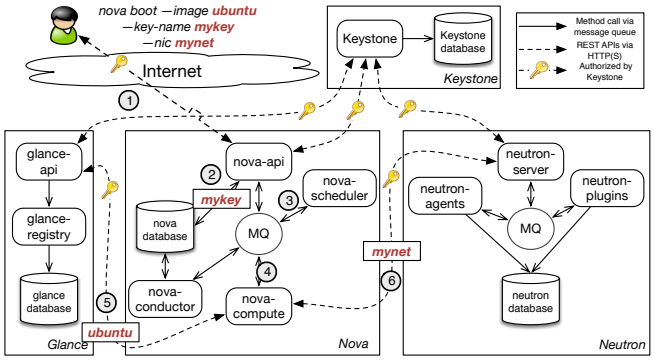

Figure 1. OpenStack processing a command to launch a VM.

Within each module, there are multiple *cloud services*. Together with other cloud services in the module, they implement tasks that the module is responsible for. For example, in the `Nova` module, a `nova-api` service parses user's commands whereas a `nova-compute` service launches and manages VMs.

Users and modules communicate using REST APIs over HTTP. Commands from cloud users and other modules are authenticated at the entry point of the module, using *user tokens*. Users supply such tokens with their commands and modules propagate these tokens to enable services to restrict the resources accessible to the tasks performed as a result of a particular user's command. Within a module, services communicate using the module's internal message queue (MQ in Figure 1), which uses an RPC mechanism rather than the REST APIs. Note that such messages are not authenticated, but user tokens are still passed into every cloud service, in case cloud services may may issue commands to different modules which requires authentication.

To better illustrate the modular design of OpenStack, we show how a user launches a new VM in OpenStack in Figure 1. In order to launch a VM, user submits a command (step 1) with her token to `nova-api` service. Each command is expressed using a set of *command options* that describe the specific cloud resources to be processed in the command. In this example, the user wants to launch a new VM using an image named `Ubuntu` with a SSH key named `mykey` and to connect the VM to a network named `mynet`. `nova-api` authenticates the user's token using the `Keystone` module. If the command is authenticated, `nova-api` retrieves the `mykey` from database (step 2), and then sends a message to `nova-scheduler` to select a cloud node to host the new VM (step 3). `nova-scheduler` picks a cloud node, and invokes the `run_instance` method of the `nova-compute` service running on the node (step 4). `nova-compute` then leverages the `Glance` (step 5) and `Neutron` (step 6) modules to retrieve the image files and setup a network connection for the VM, and finally boots up the new VM. As shown in Figure 1, after receiving commands from `nova-compute`,

both `glance-api` and `neutron-server` need to check with the `Keystone` module in order to authenticate the user token they received in order to authorize access to the resources they manage.

B. OpenStack Security Issues

While researchers are mostly concerned about security of VMs, security breaches of cloud services are very real and happen for various reasons. Researchers have identified over 100 vulnerabilities in various OpenStack cloud services [3]. At least 13 of these vulnerabilities enable remote adversaries to completely takeover a cloud node. For example, a vulnerability [10] was reported in `nova-api` service that allows remote adversaries to read/write arbitrary files on a cloud API node. As another example, a vulnerability [11] reported in a `Glance` service allows remote adversaries to execute arbitrary code on a cloud image server node. Since cloud services often run with privileges on their hosts, remote adversaries can leverage cloud service vulnerabilities to easily gain complete control over a cloud node.

In addition, insiders play an important role in cloud node breach. As a large system with potentially thousands of nodes, cloud management relies on a large crew of administrators. As much as we would like to trust them, rogue employees are indeed one of the most significant problems; many data breach incidents happen due to insider attacks [12], [13], [14].

Since cloud services are fully trusted by the cloud platform, a compromised cloud service may wreak havoc on command processing, modifying or forging messages. A compromised cloud service can modify or forge messages within a module, neither of which requires authentication or a user token. For example, a compromised cloud service can replace user-selected cloud resource for upload in Figure 1 with an adversary-chosen resource instead to make the user VM run misconfigured images, install adversary-chosen SSH keys (i.e., so an adversary may login to a user's VM), etc. In addition, a compromised cloud service may be able to forge messages entirely. For example, any compromised `nova` service (see Figure 1) may request a snapshot of any running VM owned by any cloud user, therefore obtaining sensitive data contained in the VM by forging a single message.

C. Problem Definition

In this work, we want to build a system that detects attacks against user commands. To do this, we assume that the cloud node on which the user VM runs has an uncompromised trusted computing base (i.e., hypervisor and VM host). In addition, we also trust a service for installing the cloud node software securely, such as a *cloud verifier* [15], [16].

On the other hand, we do not trust any cloud service software on the compute node or elsewhere in the cloud. In addition, we assume that it is possible that some cloud nodes, other than the one running the user VM targeted in the command, may be compromised. Thus, we offer the guarantee that as long as the cloud node running a user VM is not compromised, then that node enforces security policies that restrict how commands for that node's user VMs are executed, detecting unauthorized behaviors. Note that we do not trust the user completely. The user supplies commands, which we aim to execute as specified by the user, but those commands must be limited to the users' VMs.

Prior research has focused primarily on defenses to attacks originating from code on the same cloud node [6]. In this paper, we explore attacks against user commands that may be launched from other cloud nodes. Researchers have highlighted the danger of using publicly-available images for their user VMs, because these images may be misconfigured or maliciously configured [7]. However, this work does not address the problem of an active attacker modifying messages to change the image or modifying the image storage maliciously. Other work has explored how users may set requirements to govern when their data are released [8]. That work has a similar flavor to this work, as users define requirements to be enforced, although these requirements only govern the release of data, not the REST API broadly.

We want a solution whereby it is possible to generate a security policy that restricts the way that commands may be executed on the cloud node running the user VM to detect attacks against command execution. We envision the following requirements for such a solution.

- Prevent many possible attacks on command execution.
- Not require changes to the cloud services.
- Be possible to deploy without creating false positives.
- Have minimal impact on cloud performance.

In this work, we aim for a solution that prevents as many attacks as possible without creating false positives. It is imperative that the solution not produce false positives, as these will prevent useful work. Thus, we may not block all possible attacks. In the evaluation, we will examine quantitatively how the adversaries' attack options are reduced.

III. Solution Overview

In this paper, we explore use of a state machine security model for detecting attacks against the execution of the cloud's user commands. This choice is inspired by prior work on host intrusion detection systems (HIDS) [17], [18], [19] and our insight that we can address some limitations of this prior work effectively.

Host intrusion detection systems build state machine models of programs in order to limit the program's execution to expected sequences of system calls. We adopt the same idea to enforce legal sequences of system calls executed by cloud services for each cloud command. However, researchers found state machines for system call sequences inadequate for HIDS because they did not predict the argument values used in the system calls, leading to opportunities for mimicry attacks [20]. Since user commands mainly use cloud services

to deliver cloud resources to VMs, we hypothesize that can predict such system call argument values. Therefore, we propose to augment the state machine model to restrict the values of arguments used in system calls.

In this paper, we investigate whether we can generate and enforce *command automata* representing legal system call sequences and argument values for user commands to enforce a two-stage defense on each system call invoked by cloud services. First, the idea is that given each user command we can predict the possible sequences of system calls on the cloud node that will be invoked by cloud services to execute that command. We note that cloud services often invoke other processes to implement the command, which we will call *helper processes*. Second, the argument values to each system call will either be predicted from the command itself or restricted to safe values. Because we restrict the arguments to helper processes, we can validate their execution complies with the user command and do not need to track the system calls of these processes. By enforcing command automata, we aim to significantly restrict the adversaries' attack options without blocking authorized operations.

The command automata must prescribe the sequences of system calls perform safe execution of user commands. While cloud services perform many system calls, we make the observation that only those system calls that cloud services perform that access security-sensitive resources need to be authorized. One insight is that a security-sensitive system call must access resources belonging to user VMs, which can be detected using the host's access control policy (e.g., sVirt [21]). If all these system calls are mediated and authorized, then only approved system calls may impact user VM resources. To ensure protection of the host and helper processes, however, we sandbox the cloud service system calls that do not impact the user VM; if a command does not impact a user VM it need not be run in the host VM.

The command automata will be augmented with *argument constraints* to limit the values of arguments that will be accepted in automata transitions. Our insight is that many argument values can be predicted from the command itself and that these predictions can be enforced on cloud service system calls. For the few arguments values that cannot be predicted in advance, we find that we can prescribe constraints to limit the system call to values that are safe for the user VMs. We describe how we use dynamic analysis results to prepare templates for predicting the values of arguments in Section IV-B.

The command automata are enforced by the CloudArmor system described in Section V-B. CloudArmor includes a *user proxy* that obtains argument value predictions from user commands for instantiating the argument constraints, described in SectionV-A. CloudArmor sandboxes cloud services to protect VM hosts and enforces the command automata on system call executions as described in Section V-B.

IV. Command Automata Construction

We leverage dynamic analysis to construct state machine models for the user commands. To collect traces, we propose to configure a "training cloud" that is configured in the same manner as the deployment cloud and generate command instances to provoke the cloud services into processing these commands (i.e., generating system calls for collection). We require the same setup of the training and deployment cloud in order to precisely model how deployment cloud would process user commands. If any update is made to the deployment cloud (e.g., code, configuration), regression testing would be used to detect if the update changes the traces, thus requiring new training. To generate command instances, we identify all the command options and dependencies between command options (i.e., those required or prohibited by an option) to generate command instances that apply the legal sets of command options.

The two challenges in this task are to convert the collected traces into command automata and to extract argument constraints from the traces to annotate the command automata.

A. Constructing Command Automata

Given a trace set for a particular command, we compute the command automata that describes the system call sequences of cloud services when serving the command. The challenge is to make the state machine *precise*, such that spurious system call sequences which may enable mimicry attacks are not legal in the automata.

We leverage an existing tool, called Synoptic [22], that builds nondeterministic finite state machines (NFA) from system call traces by *merging* the trace set to satisfy three types of temporal invariants: always followed by, never followed by, and always precedes. Using these invariants, Synoptic eliminates impossible paths from the NFA, which removes many of the cases where executions not found in the traces would have been accepted by the NFA.

By default, we build a command automata for each command, feeding the traces for that command into Synoptic. However, different combinations of options for a single command may yield very different traces, so we also enable the generation of multiple command automata per command. That is, command automata may be associated with a command and a set of options.

Take the `nova reboot` command as an example, if the option `--hard` is used, instead of rebooting, a VM will be destroyed and re-created, causing a very different sequence of system calls than the default. In this case, we would like to detect from the system call sequence if adversary has hijacked the request and changed users reboot option. However, it is not possible to do so if we have a single automaton for all `nova reboot` commands.

The solution we have adopted is to cluster traces for command options if the traces differ by no more than a

threshold number of system calls. We then produce command automata for the clusters of command options that cause significant differences in system call sequences. In this way, we preserve the ability to prevent an adversary from modifying command execution undetected without introducing a large number of automata.

We cannot claim that the state machines constructed are precise—they may still contain spurious system call sequences. In Section VI-A we will evaluate the precision of our command automata and their ability to limit the attack options available for adversaries.

B. Identifying Argument Constraints

In this task, we use the traces to identify argument constraints for the system calls in the automata transitions. There are three classes of arguments we want to identify: (1) those that are dependent on user input; (2) those that are constants; and (3) those that are generated by the cloud. For the first class, constraints can be instantiated once a user command is specified at runtime. For the second class, the values can simply be fixed in the command automata. For the third class, our goal is to restrict these argument values to ones that are safe for cloud platform and the user VM targeted by the command.

Given a set of traces for a generated command automata, we propose to detect dependency between the command's options and the system call arguments. To detect dependence between a specific option and system call arguments in the trace set, we detect the argument values that change when only that specific option's value is changed. We aggregate the sets of traces that have the same values for all but one common option. We then examine those traces to identify the system call argument values that changed in each of those traces. To make this systematic, we order the traces for a specific command automata in shortlex order.

After removing the set of arguments that are dependent on command input, we either have fixed arguments or arguments that are defined by the cloud services. Fixed arguments have the same value for each trace in the trace set for the corresponding system call in the automata. We find that a large percentage of the system call argument values are fixed by the cloud configuration.

The remaining argument values are generated by the cloud services themselves. Many of these include files and directories created by cloud services to store data temporarily on the VM host. These argument values are relatively ad hoc, so we produce argument constraints manually to prevent using another user VMs' data and overwriting of other data belonging to this user VM. Fortunately, we find that the number of ad hoc argument constraints is modest in practice. Note that these ad hoc constraints may not prevent all possible attacks against the user VM, but we can isolate these arguments from other user VMs.

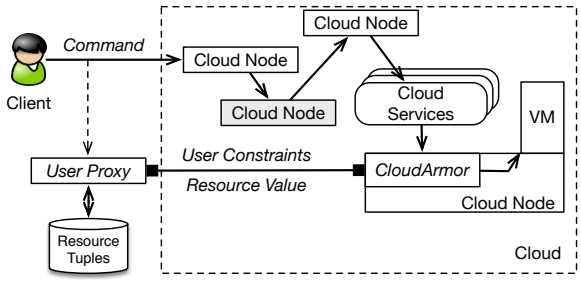

Figure 2. An overview of CloudArmor enforcement framework.

V. Command Automaton Enforcement

An overview of the enforcement framework is shown in Figure 2. When user sends a command, *user proxy* parses the command and sets the argument constraints. The user proxy uses a *resource tuples* store to map names of resources to values. Then argument constraints are sent to the *CloudArmor module* in cloud via a secure channel [16][1]. The CloudArmor module then instantiates a command automaton for the command, and starts authorizing system calls from cloud services. A system call is allowed to run on the cloud node if and only if it can cause a state transition of the command automaton (i.e., fall into an order allowed by automaton and arguments satisfy constraints). Upon detecting unauthorized system calls, the CloudArmor module aborts the execution of cloud services and raises an alarm. If new cloud resources were created as a result of an authorized system call, the CloudArmor module returns its value to user proxy via the secure channel.

A. User Proxy

Instantiation of command automata requires predicting values for argument constraints. These constraint values are collected by the *user proxy* on users' local machines. Given a command and its options, the user proxy will automatically retrieve the expected values for these option values, and send those constraints to the CloudArmor module via a secure channel for instantiating the command automata.

The challenge for the design of the user proxy is to predict values for the system call arguments from the options in each user command. Fortunately, we find that there exists a simple mapping between the two. Command options fall into two categories, names and values. Names are references to resources that are stored in cloud (e.g., image_id, key_id). When serving a command, cloud services resolve these names to actual values and then use them as arguments in system calls. Values are often used as-is by cloud services in system calls. Consequently, producing user constraints simply boils down to predicting the values of the cloud resources referenced in each user command.

[1]We leverage an existing approach [16] to establish the secure channel between user proxy and the CloudArmor module. The secure channel should be able to prevent adversary controlled nodes from modifying the user constraints or replaying them. Refer to the paper [16] for more details.

The idea here is that user proxy would store and maintain name-value mappings of cloud resources at runtime. We refer these mappings as *resource tuples*, $resource = (type, name, value)$, with the type being the type of resource (e.g., key, image, etc.). However, due to the existence of adversary-controlled nodes, the user proxy cannot retrieve trustworthy values of cloud resources from the cloud.

Through investigating different kinds of cloud resources referenced in OpenStack commands, we find that there are three ways resource tuples are created. First, they are created when users store resources in cloud. For example, user creates SSH keys and stores them in cloud by sending the *nova keypair-add* command. In this case, user proxy would create a resource tuple recording the name-value mapping of the key being created. Second, if the cloud resources are created within the cloud (e.g., VMs), the CloudArmor module would act on behalf of user proxy and return the value of the resource to the user proxy. Third, users may manually create resource tuples after verifying that the values of resources meet their expectations. For example, user may reference a third-party image in her command, after she verifies the checksum value of the image and stores the mapping to the user proxy.

B. CloudArmor Module

CloudArmor module runs as part of the trusted computing based on cloud nodes[2]. It intercepts system calls issued by cloud services, and authorizes them against the instantiated command automata.

Since command automata only model a subset of system calls issued by cloud services (See Section III), the CloudArmor module needs to make a decision on what to do with the rest. One choice is to let them execute freely on the cloud node, since they are not directly affecting users' VMs (i.e., not accessing labeled resource). However, such design introduces a loophole that enables adversaries to indirectly affect users' VMs by launching local exploits. For example, assume cloud services write an unlabeled file which is not relevant to a user VM. Later if the same file is accessed by a helper program which cloud services use to modify a VM, adversary can exploit vulnerabilities in the helper program to launch attacks against the VM indirectly. While we could extend our command automata to authorize these system calls as well, it introduces a great deal of complexity that would lead to imprecise model and enable mimicry attacks. So the design choice we made is to sandbox cloud services.

The CloudArmor module creates a sandbox environment that is sufficient to run the cloud services (e.g., a virtual machine or container). If a system call issued by a cloud service targets a labeled resource or helper process, the CloudArmor module authorizes the system call and then forwards it to the cloud node for execution. Other system calls such as loading libraries, establishing connections to message queues, are executed within the sandbox, ensuring that they cannot affect either the users' VMs or the cloud node itself. We leveraged existing mechanisms for system call forwarding (e.g., Proxos [23] and Prospect [24]). While such forwarding may incur non-trivial overhead, they are not on the critical path of VM execution (See Section VI-C).

[2]In this work, we focus on compute nodes since they host users' core asset—VMs, but our mechanism can be easily extended to other kinds of cloud nodes such as API node, Neutron server node , etc..

Table I
AUTOMATON METRICS

	Automaton		Argument Constraints				Average
	States	Trans.	User	Built-in	Cloud	Predictable(%)	Branching Factor
boot	42	48	6	140	19	89%	1.14
delete	8	8	3	15	0	100%	1.0
image-create	4	6	2	10	3	80%	1.5
add-secgroup	11	13	3	21	0	100%	1.18
rescue	34	38	5	121	25	83%	1.12
rebuild	46	56	7	158	27	85%	1.22
migrate	81	95	8	193	26	89%	1.17

VI. EVALUATION

A. Automaton Complexity and Precision

In this section, we quantitatively evaluate the restrictive power of command automata. We evaluate both the precision of the automata and the percentage of system call arguments that are constrained. In an ideal case, every user command would result in an unique sequence of system calls with pre-determined sets of arguments being executed on a cloud node. Consequently, adversary-controlled nodes cannot launch *any* attack, since any deviation in the system calls would be detected.

We have evaluated user commands for 55 types of commands in the OpenStack Grizzly release. Due to space constraints, we only show seven most frequently used ones. To evaluate automata precision, we use Wagner's average branching factor metric [17]. This metric measures the average number of possible transitions after each state in each automaton. Smaller numbers are more favorable as adversaries are afforded less freedom for crafting a malicious system call sequence without deviating from the model. Table I shows the average branching factor for the seven most-complex OpenStack commands. Our observation is that command automata are far simpler than the model of general programs (e.g., `sendmail` has an average branching factor larger than 15 [17]); their average branching factors are close to 1. What this means is that the command automata describe the behavior of cloud services when serving users' commands closely; attackers are less likely to launch attacks without being detected by the automata.

Another interesting fact is the complexity of the automata. In practice, less complex automata (i.e., fewer states and transitions) are often more precise. We demonstrate the complexity of automata by showing the number of states and transitions in each automaton. Of all 55 commands, `migrate` is the most complex, and it has only 81 states and 95 transitions. Automata are simple because they are

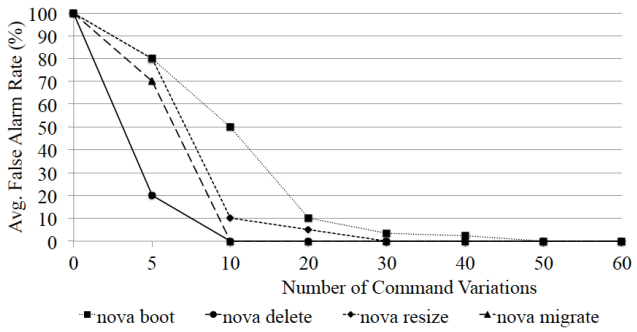

Figure 3. False alarm rate.

constructed using only security-sensitive system calls and system calls that invoke helper processes.

Next we evaluate how many system call arguments can be predicted and constrained. We classify system call arguments into three categories: the first category represents user-specified arguments, whose values are dependent on user's command, the second category represent constant arguments, and the third category represent arguments constrained by the cloud environment. Argument values of the first two categories can be predicted exactly at command automata instantiation time. Argument values for the third category can be restricted to satisfy specific set, range, or filesystem subtree constraints. The number of system call arguments in each category is shown in Table I. As shown in the table, most of the system call arguments are in the second category (an average of 86%). This is due to the fact that cloud services interact with their host nodes mostly by invoking helper processes, and the way helper processes are invoked is fixed for the configuration. User-specified constraints account only for 4% of the system call arguments, although they are a critical set. Both constant and user-specified constraints are known at command instantiation, so on average 90% of the system call arguments are predictable. As a consequence, adversaries can only affect the remaining 10% of the system call arguments to launch attacks. During the experiments, we find that these arguments are temporary files (e.g., temporary directory to mount a VM disk) and file descriptors (e.g., socket to libvirt) which can be easily constrained using cloud constraints.

B. False Alarms

We evaluate the false alarm rate with respect to the number of traces generated to build the command automaton. Results are shown in Figure 3, which shows the false positive rate relative to the number of training commands run. We only plot data corresponding to the four most frequently used commands (`nova boot`, `nova delete`, `nova resize`, and `nova migrate`). In this experiment, we generate commands by varying command options and their values among a set of legal combinations. We then run the commands and determine whether command automata constructed up to that point would authorize the command, where blocking the command would result in a false positive.

Table II
VM PERFORMANCE OVERHEAD.

	Vanilla	CloudArmor	Slowdown
Macro benchmark			
KBuild (sec)	70.7 ± 0.1	71.1 ± 0.2	0.5%
Apache (req/s)	8727 ± 325	8654 ± 296	0.8%
dbench (MB/s)	323.5 ± 1.7	322.3 ± 2.1	0.4%
Lmbench			
simple syscall	0.1165	0.1167	0.2%
read	0.2576	0.2607	1.2%
write	0.2269	0.2279	0.4%
open/close	1.1323	1.1338	0.2%
fork + execv	177.967	180.233	1.2%

That is, at run of command n, we examine whether that command would be authorized by a command automata produced from the previous $n-1$ commands.

As we can observe from the figure, three of the commands register a 0% false alarm rate as soon as we train the model with 30 variations of the command. The only command that requires a higher number of command variations is the `nova boot` command, which takes about 50 runs. The reason is because `nova boot` has several command options and can be invoked in several ways, due to the high number of optional arguments.

C. Performance

We evaluate the performance of our framework by measuring the impact on users' VMs. Our experimental setup involves: (1) a vanilla OpenStack infrastructure and (2) a OpenStack infrastructure with CloudArmor enabled. We use OpenStack Grizzly with KVM. User VMs are configured with 2 VCPUs, 4 GB memory, and 10 GB virtual disk. Each cloud node only hosts a single user VM. The following benchmarks are used to measure the performance of VM: (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; and (3) dbench for disk I/O intensive workloads, where we measured filesystem I/O throughput (MB/sec) with 20 concurrent clients. For microbenchmarks, we used lmbench. The results are shown in Table II. For various benchmarks, the maximum slowdown for a VM is less than 1.2%. The overhead is relatively small because CloudArmor is not intercepting system calls from users VMs. Instead, it only authorizes system calls issued by cloud services as a result of user commands.

VII. RELATED WORK

A line of research has investigated guaranteeing the integrity of cloud resources. Techniques like Accountable VMs [25] and Trinc [26] 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.

Research in hardening hypervisors from both co-resident VMs and insiders has become popular. Approaches like NoHype [27] minimize the VMM's attack surface by eliminating all but a small resource management kernel. Other techniques like formally assured L4 microkernels [28], CloudVisor [5] and privileged domain separation [23] again limit the impact a compromised hypervisor or privileged domain can have on hosted VMs. Finally, low-level monitors like DeepSafe [29] 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.

CloudArmor framework shares many ideas with host-based intrusion detection systems. Host-based IDS builds a behavior model of program via static [17], [18], [30] or dynamic [31] 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 user VMs. This greatly reduces the model complexity in CloudArmor and enables argument value prediction.

VIII. Conclusion

The growing popularity of cloud platforms has resulted in the need for secure cloud platforms. However, current cloud platforms have a bloated trusted computing base (TCB), where compromise of any single cloud node may result in the compromise of the whole cloud platform. In this paper, we focus on protecting the execution of users' cloud commands from attack by compromised cloud services. To defend against these attacks, we present CloudArmor, a framework that restricts operations performed on cloud nodes using a state machine model of command execution derived from runtime analysis. We implement CloudArmor for the widely-used OpenStack cloud platform, and show it can block attacks without incurring false alarms for less than 1.2% performance overhead on user VMs.

References

[1] "Amazon Simple Storage Service(Amazon S3)," http://aws.amazon.com/.

[2] "Rackspace Cloud Servers," http://www.rackspace.com/cloud/.

[3] "Openstack vulnerabilities," http://www.cvedetails.com/vulnerability-list/vendor_id-11727/Openstack.html.

[4] Y. Sun, G. Petracca, and T. Jaeger, "Inevitable failure: The flawed trust assumption in the cloud," in *CCSW '14*.

[5] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *SOSP'11*.

[6] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy, "Self-service cloud computing," in *CCS '12*.

[7] S. Bugiel, S. Nürnberger, T. Pöppelmann, A.-R. Sadeghi, and T. Schneider, in *CCS '11*.

[8] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-sealed data: A new abstraction for building trusted cloud services," in *USENIX Security '12*.

[9] "Home - OpenStack Open Source Cloud Computing Software," http://www.openstack.org/.

[10] "CVE-2013-1665," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1665.

[11] "CVE-2014-0162," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0162.

[12] F. Rocha and M. Correia, "Lucy in the sky without diamonds: Stealing confidential data in the cloud," in *DSNW '11*.

[13] "Google fires engineer for privacy breach," http://www.cnn.com/2010/TECH/web/09/15/google.privacy.firing/.

[14] "Statistics — DataLossDB," http://datalossdb.org/statistics.

[15] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel, "Seeding Clouds with Trust Anchors," in *CCSW '10*.

[16] J. Schiffman, Y. Sun, H. Vijayakumar, and T. Jaeger, "Cloud verifier: Verifiable auditing service for IaaS clouds," in *SERVICE '13*.

[17] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *IEEE S & P '01*.

[18] J. T. Giffin, S. Jha, and B. P. Miller, "Detecting manipulated remote call streams," in *USENIX Security '02*.

[19] W. Lee and S. J. Stolfo, "Data mining approaches for intrusion detection," in *Usenix Security*, 1998.

[20] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *ACM CCS '02*.

[21] "SVirt Project," http://selinuxproject.org/page/SVirt.

[22] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *FSE '11*.

[23] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: making trust between applications and operating systems configurable," in *OSDI '07*.

[24] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: peripheral proxying supported embedded code testing," in *AsiaCCS '14*.

[25] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, "Accountable virtual machines," in *OSDI '10*.

[26] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "Trinc: small trusted hardware for large distributed systems," in *NSDI '09*.

[27] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *ACM CCS '11*.

[28] G. Klein *et al.*, "seL4: Formal Verification of an OS Kernel," in *SOSP '09*, 2009.

[29] "http://theinvisiblethings.blogspot.com/2012/01/thoughts-on-deepsafe.html," http://www.mcafee.com/us/solutions/mcafee-deepsafe.aspx.

[30] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller, "Environment-sensitive intrusion detection," in *RAID '05*.

[31] A. Somayaji and S. Forrest, "Automated response using system-call delays," in *USENIX Security '00*.