Elite: Automatic Orchestration of Elastic Detection Services to Secure Cloud Hosting

Yangyi Chen[1(✉)], Vincent Bindschaedler[2], XiaoFeng Wang[1], Stefan Berger[3], and Dimitrios Pendarakis[3]

[1] Indiana University Bloomington, Bloomington, USA
{yangchen,xw7}@indiana.edu
[2] University of Illinois Urbana-Champaign, Champaign, USA
bindsch2@illinois.edu
[3] IBM Thomas J. Watson Research Center, Yorktown Heights, USA
{stefanb,dimitris}@us.ibm.com

Abstract. Intrusion detection on today's cloud is challenging: a user's application is automatically deployed through new *cloud orchestration* tools (e.g., OpenStack Heat, Amazon CloudFormation, etc.), and its computing resources (i.e., virtual machine instances) come and go dynamically during its runtime, depending on its workloads and configurations. Under such a dynamic environment, a centralized detection service needs to keep track of the state of the whole deployment (a *cloud stack*), size up and down its own computing power and dynamically allocate its existing resources and configure new resources to catch up with what happens in the application. Particularly in the case of anomaly detection, new application instances created at runtime are expected to be protected instantly, without going through conventional profile learning, which disrupts the operations of the application.

To address those challenges, we developed *Elite*, a new elastic computing framework, to support high-performance detection services on the cloud. Our techniques are designed to be fully integrated into today's cloud orchestration mechanisms, allowing an o rdinary cloud user to requ est a detection service and specify its parameters conveniently, through the cloud-formation file she submits for deploying her application. Such a detection service is supported by a high-performance stream-processing engine, and optimized for concurrent analysis of a large amount of data streamed from application instances and automatic adaptation to different computing scales. It is linked to the cloud orchestration engine through a communication mechanism, which provides the runtime information of the application (e.g., the types of new instances created) necessary for the service to dynamically configure its resources. To avoid profile learning, we further studied a set of techniques that enable reuse of normal behavior profiles across different instances within one user's cloud stack, and across different users (in a privacy-preserving way). We evaluated our implementation of Elite on popular web applications deployed over 60 instances. Our study shows that Elite efficiently shares profiles without losing their accuracy and effectively handles dynamic, intensive workloads incurred by these applications.

弹性计算框架，支持高性能检测服务。

允许原生云用户发出请求检测服务，从而方便的确定相关参数。

对大量的数据流进行并行分析

H. Bos et al. (Eds.): RAID 2015, LNCS 9404, pp. 588–614, 2015.
DOI: 10.1007/978-3-319-26362-5_27

1 Introduction

Cloud computing has emerged as the mainstay of cost-effective, high-performance platforms for hosting personal and organizational information assets and computing tasks. This thriving computing paradigm, however, also comes with security perils. Recent years have seen a rising trend of security breaches in the cloud. Examples include the high-impact attacks on Amazon AWS [1] and Dropbox [2]. Countering those threats requires effective security protection, which today mainly relies upon the virtual-machine (VM) instances with assorted protection mechanisms (firewalls, intrusion detection systems, malware scanners, etc.) pre-installed. Prominent examples include McAfee, Trend-Micro and Alert Logic [3–5], all of which provide such instances with embedded security agents to their customers for running their cloud applications.

A fundamental issue for this solution is that each instance has to allocate resources for accommodating a complete security system, even when it cannot make full use of the system most of the time. This goes against the resource-sharing, on-demand service design of the cloud. As a result, the same security functionalities are duplicated across all the instances when running a cloud application, even for those temporarily rented to handle the burst of workloads. Resources are squandered in this way when some instances do not have enough workloads to merit the cost of operating the whole set of security mechanisms (e.g., loading a large number of signatures into memory for malware scanning). This inevitably interferes with the operations of the user application running within the same instance. Further, security systems hosted in different instances work independently, which fails to leverage cross-instance information to better protect the cloud.

All these problems can be addressed by a centralized detection framework that concurrently serves different application instances, for example, through inspecting audit trails from those instances to detect cloud-wide malicious activities. Deployment of such a service, however, faces significant challenges in a dynamic cloud environment, where different instances are generated and their connections with assorted computing resources (storages, other instances) are established during runtime. Management of this environment is complicated enough to justify the use of an *orchestration* mechanism, such as OpenStack *Heat* [6] and Amazon *CloudFormation* [7], which automatically creates different types of cloud instances in response to changes to an application's workload, links them to each other and other cloud resources, and further configures the *software stack* (a set of software packages for performing a task) within individual instances according to the cloud user's specifications (called *template*). Clearly, to protect those instances, the detection service needs to work closely with the orchestration process.

Challenges. To better understand the challenges for the detection service to work in the cloud environment, let us look at a simple system with three *auto-scaling* groups, whose resource level, such as the number of instances, goes up or down automatically according to conditions set by the cloud user. These groups

are assigned to host load balancers, web servers, and database servers respectively. Effective protection of all their instances needs a high-performance detection engine capable of auto-scaling. More specifically, during runtime, some of the groups may expand, adding in more instances to help manage extra workload. This puts the detection service protecting them under pressure, which may also need to scale up its capability to handle new tasks so that it can timely respond to malicious events. This process is actually much more complicated than it appears to be, involving re-arrangement of tasks, for example, connecting a new detector instance to a set of application instances, and configuring the detector properly according to the contexts of those instances, e.g., whether they are load balancers, web servers or database servers.

Further, in the case of anomaly detection, where the intrusion detection system (IDS) is supposed to catch an application instance's deviation from its normal behavior, the detector needs to instantly create a normal behavior profile for every instance generated in an auto-scaling process, based upon its context information such as its software stack and configurations. Note that this needs to be done in the most efficient way possible: given the large scale of the computation the cloud undertakes and timely responses it needs to make to workload changes, we cannot directly adopt conventional approaches, such as profile learning, which could take a long time to build up an instance's behavior model. Those issues can only be addressed by new techniques that incorporate the detection service into the orchestration mechanism, making the service work seamlessly within the cloud work-flow and tuning it to meet the high performance demands from the cloud. However, except for some attempts to directly deploy secure instances (just like ordinary instances) through CloudFormation, as TrendMicro [4] does, so far, little has been done to understand how to automatically arrange, coordinate and manage centralized security services and their resources through cloud orchestration.

Our Approach. To this end, we developed a new *elastic intrusion detection framework*, called *Elite*, as an extension of the cloud orchestration mechanism. Elite is designed to support parallel detection systems, automatically scheduling them on-demand, provisioning their resources and allocating their tasks. A cloud user can conveniently require detection services through her template when she specifies the infrastructure for her cloud application using the file. Based upon the template, an orchestration mechanism enhanced with Elite automatically creates detector instances, configures them based upon their context information, connects them to cloud resources and scales their number up and down according to the application's workload. Also as part of the framework, those instances accommodate a high-performance distributed stream-processing engine to support different detection techniques.

Elite also includes a novel technique that quickly builds up a normal profile for each newly-created application instance, through adjusting the profiles of other instances within the same auto-scaling group. The idea here is that since those instances accommodate an identical software stack with identical configurations, and are tasked to process the same datasets, their behaviors should be very

similar (but not identical). More specifically, our approach first generalizes the profiles from multiple instances within the same group into a *profile template*, and then specializes it using the parameters (e.g., temporary file names) of a new instance automatically identified. In this way, instances acquired by the cloud application during its runtime are automatically profiled and protected. We further developed a technique that enables profile reuse across different auto-scaling groups, particularly when a cloud user wants to take advantage of existing profiles (from other users) associated with a similar software configuration to avoid training her detector instance from scratch. This profile sharing needs to be done in a privacy-preserving manner, given the sensitivity of individual users' system configurations. Our approach is built upon a simple security protocol that helps one user retrieve the profile from another party when their configurations are close enough, without exposing their sensitive information to each other and the cloud.

特色：利用了heat和storm

We implemented Elite on Openstack's Heat orchestration engine and utilized *Apache Storm*, a high-performance stream processing system, to build detection services. Our implementation was evaluated using a cloud application capable of scaling up to 60 instances. During our experiments, Elite ran anomaly detection and the profiles it automatically created were found to work effectively, without any negative impact on the effectiveness of the underlying detection technique. Also, our stress test shows that Elite introduced negligible overheads and was well adapted to dynamic workloads.

Contributions. The contributions of the paper are summarized below:

- *New framework for elastic detection services.* We designed a new framework to support high-performance auto-scaling detection services. This framework has been fully integrated into the existing cloud orchestration mechanism, which enables a user to conveniently specify the parameters for detector instances together with other cloud resources she requests for her application. The cloud then acts on such specifications to automatically structure and restructure the infrastructure of the detection service and scale its capability in accordance with the application's workload. Our framework also utilizes a high-performance stream processing engine to accommodate different detection techniques.
- *Profile reuse techniques.* We developed a suite of new techniques to reuse normal behavior profiles within one auto-scaling groups and across different groups. Our approach leverages the similarity among different cloud instances' software configurations to generate accurate profiles for the instances dynamically created during runtime. Also, sharing of profiles between different users is supported by a security mechanism that preserves the privacy of the parties involved.
- *Implementation and evaluation.* We implemented our design on OpenStack and evaluated it using 60 cloud instances. Our study shows that our framework and techniques work effectively in practice, incurring negligible overheads.

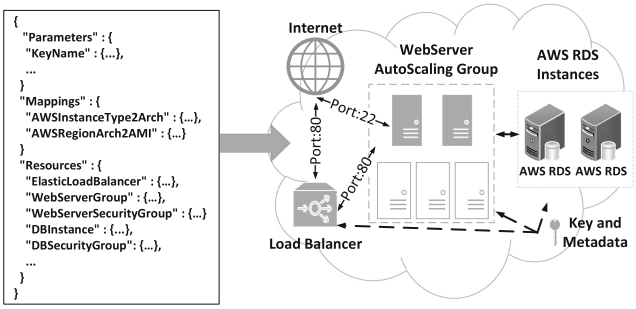

Fig. 1. Cloud orchestration example

2 Background

In this section, we provide background information for our research, including brief introductions to cloud orchestration techniques, IDS systems and the stream processing engine that supports our high-performance detection service.

Cloud Orchestration. The orchestration mechanism, also known as *orchestrator*, is a critical cloud service that automates arrangement, coordination and management of the cloud resources for a complicated application the user deploys on the cloud. To use the service, the user is supposed to describe the infrastructure of the application in a text file (a template) and submit it to the orchestrator. Within the template are the specifications of types of cloud instances, their software stacks and their relationships with other resources, for example, how an Amazon EC2 instance is connected to an Amazon RDS database instance. For each instance, one can utilize the configuration management tools integrated with the template to define how its software should be configured. Also various services provided by the orchestrator, such as auto-scaling, can also be requested as a resource within the template. Figure 1 shows a simplified template file and the application infrastructure it defines.

在template中，定义了instance的类型，软件栈，资源之间的关系。

Using the template, the orchestrator automatically manages the whole lifecycle of the user's application, acquiring new resources from the cloud, configuring them, re-arranging tasks for different instances when new workloads come up and deleting resources when a job is done. The whole infrastructure can be conveniently adjusted by the user through updating her template. Such functionalities are supported by mainstream orchestration products, including Openstack Heat [6] and Amazon CloudFormation [7]. Also well-specified templates are extensively reused and customized by cloud users to quickly deploy their applications.

In our research, we incorporate a high-performance parallel IDS service into the orchestration mechanism as a cloud resource, which can be conveniently specified in a template. To provide the service with context information necessary for its operations, we further modified the orchestrator, adding a new channel for it to communicate with the IDS resource. Also, we constructed a template with necessary configurations to enable auto-scaling and configuration of detector

将IDS服务作为一种资源，集成到了编排机制中。用户可以方便在template中指定使用该服务。

instances, which can be customized by a cloud user to integrate the service into her application infrastructure.

Intrusion Detection. Intrusion detection techniques have served as the backbone of organizational security protection for decades. An IDS detects malicious activities either through identifying a set of signatures (signature-based detection) or through monitoring a system's deviations from its normal behavior profile (anomaly detection). Examples for the former include network intrusion detectors like snort, which screens network traffic flows for the patterns of known threats (e.g., propagation of Internet worms), and host-based scanners that inspect the code of suspicious programs or their behaviors (e.g., system calls) [8–11] to catch malware. Anomaly detection, on the other hand, typically looks at a legitimate program's operations to find out whether it is doing something that it normally does not do, based on its behavior profile. Such a profile can be system-call sequences [12–14], or just a white list of system calls (with parameters) the program is supposed to make when it is not compromised, as many host-based detectors (e.g., *systrace* [15]) do. In our research, we implemented a simple signature-based detector and a white-list based anomaly detector for inspecting audit trails submitted by application instances. Note that our framework is also capable of supporting more complicated techniques, like call-sequence based anomaly detection.

Such conventional detection techniques, particularly those host-based, are not designed to serve a large number of cloud instances and process a large amount of data at a high speed. To move them into a cloud service, we need to incorporate them into a parallel, high-performance computing platform. What was adopted in our design is a stream processing engine, as elaborated below.

Stream Processing. A stream processing engine is a distributed computing system for analyzing unbounded streams of data in real time. This capability is crucial to the mission of the cloud detection service, which receives a large amount of data streamed from different instances. Examples of such systems include IBM InfoSphere Streams [16], Apache Storm [17] etc. In our research, we built our distributed detection service on top of Storm.

Storm is an open-source system known for its fast speed and ease of use [18]. A typical Storm system is deployed as a *cluster*, which includes a *Nimbus node*, a *Zookeeper* [19] node and a set of *Supervisor* nodes. The Nimbus node is the master of the whole cluster, in charge of managing the interaction topology of the cluster as well as task allocation and tracking. Zookeeper helps coordinate Nimbus and Supervisors, which run a group of *worker* processes to do the real job. Among the workers, a set of *sprout* processes receive streaming data from other cloud instances and route it to *bolts*, which perform the user's computation task on the data (e.g., filtering, aggregating, database access, etc.). The interconnections among sprouts and bolts form a topology managed by the Nimbus and predefined by the Storm user.

Over the Storm platform, we specified a topology and implemented detection algorithms into bolts, together with a mechanism for interacting with the Heat

engine for getting context information of newly-created instances. The details of this parallel detection service are explicated in Sect. 3.2.

3 Design and Implementation

In this section, we present the design of Elite, which is meant to achieve the following goals:

何尝不是安全迁移时的目标

- *High performance*. The detection service should work in parallel, concurrently processing a large number of streams, and also auto-scaling, dynamically extending and shrinking its computing resources in response to changes to its workload.
- *Context information support*. The framework should be able to effectively communicate with the detection service a cloud application's state information. Particularly when new resources are added into the application's infrastructure, the detection service needs their context information to determine how to protect the resources.
- *Ease of use*. Through our framework, we expect that a cloud user can directly require the detection service and set its parameters, including the amount of computing resources she is willing to rent for the service and detection algorithms, without going through a complicated process of configuring individual instances that host the service.

用户可以直接请求探测服务并且设置相应的参数，例如探测算法。给我的启发：用户在迁移时，能够自定义对迁移目的主机的安全要求，以及迁移过程中的安全等级。

This design was implemented over the open-source OpenStack Heat orchestration system and the Apache Storm engine. Here we first describe Elite at a high level and then elaborate on its technical details.

详尽叙述

3.1 Overview

Architecture. The architecture of Elite is illustrated in Fig. 2. This detection framework has been built around the cloud orchestration system. It includes an extension to the *cloud formation language* used to describe an application's infrastructure for automatic configuration of our detection service, a communication mechanism built into the orchestration system for collecting the contexts of the application's runtime and dispatching such information to the detector instances, and a set of VM images for different types of parallel detection mechanisms (signature-based or anomaly detection) implemented over a stream-processing engine.

How it Works. Consider a cloud user who requests a detection service from the cloud to protect her application. All she needs to do is to state the detection image (associated with different detection techniques) and auto-scaling conditions (e.g., creating a new instance when the processing time of the detection service goes above a threshold) within the template she submits to the cloud for running her application. The specifications are then parsed by the orchestration system, which runs a template we built to automatically configure the whole

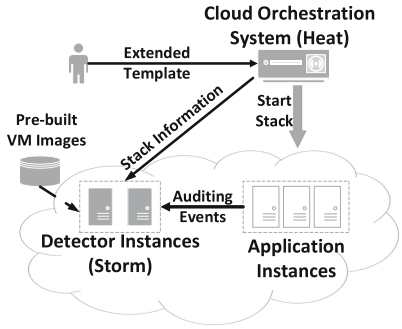

Fig. 2. System architecture

detection infrastructure, setting parameters for individual detector instances and connecting them to application instances. During the application's runtime, the auditing daemons(which can be pre-configured and incorporated into the application image) within its instances stream audit trails to the detectors. Also, the orchestrator continuously updates the detectors the state of the application, particularly context information of any auto-scaling groups or new instances dynamically generated. In the case of anomaly detection, the orchestrator can run a few "training" instances that generate input traffic to application instances for building up their normal behavior profiles and bootstrapping a whole auto-scaling group. When this happens, the detector needs to be informed that the system operates in a "training mode[1]".

All the detector instances are created from a selected detection image. They work concurrently on a large number of streams from application instances. Those detectors can go through all the audit trails to look for attack signatures or any deviations from the application's normal behaviors. When it comes to anomaly detection, the detectors also need to find out whether the system runs in a training mode, in which they automatically build up profiles for different instances (e.g., a list of system calls and their parameters) and further generalize them into a profile template within each auto-scaling group. During the normal operation of the application, such profile templates are specialized automatically to provide instant protection for every new instance. Also, profiles are reused across different auto-scaling groups and even different users to shorten or even remove the whole training stage. Such reuse needs to happen in a privacy-preserving way, given the sensitivity of the user's system configurations. This is achieved in our system through a simple security mechanism.

[1] How long the detector needs to stay in "training mode" depends on many factors such as the nature of the service provided by the application instances, the quality of training inputs, and to what extent the cloud user can tolerate the false positives. Precise tuning of the training time and the trade-offs involved is not the focus of this paper.

Adversary Model. What we built is a high-performance, auto-scaling cloud platform for supporting IDS. Depending on specific IDS techniques running on top of Elite, we need to make different assumptions about the adversary's capabilities. Specifically, for a normal host-based IDS, the sensor for collecting audit trails runs as an OS daemon. In this case, we have to assume that the OS kernel is sound. On the other hand, if the sensor is deployed at the level of virtual machine monitor, the detection system operating within Elite can catch kernel-level attacks. Also, we consider that the orchestrator has not been compromised and the cloud is honest but curious when it comes to cloud users' data privacy. In practice, commercial cloud providers tend to refrain from inspecting the content of their customers' instances for liability concerns. All we want to avoid here is to expose more data than the customer is willing to share to the cloud.

3.2 Detection Service Orchestration

As described before, Elite supports a convenient set-up of a high-performance detection service and automatic orchestration of the service in response to the states of the user's cloud application. Here we elaborate the techniques behind this elastic, scalable detection platform, including the extension made to the template language, a stream-based IDS platform and a high-performance detection system built on top of it, and the mechanism for coordinating the cloud orchestrator and our detector instances.

Detection Service Specification. To integrate our detection service into the cloud orchestrator, we extended the template language to allow the user to conveniently set up the detection service. Specifically, our extension includes a new group of cloud resources "`AWS::IDS::ENGINE_NAME`", which describes a special auto-scaling group for detectors, whose type (`ENGINE_NAME`) is specified by the user. Each detector type is associated with a pre-built VM image that hosts a parallel, stream-based detection algorithm. As an example, we implemented in our research a concurrent anomaly detector running Systrace-like profile based detection [15]. The detector was constructed in a way that it can easily incorporate other detection mechanisms and also support attack path analysis, which we discuss later.

Using this new statement, the user can request the detection service from the cloud and further describes auto-scaling conditions for the service. For example, the user can specify a fixed auto-scaling ratio or let the detection service auto-scale based on conditions like `message processing delay` or `CPU usage` of the detector instances. After Heat parses the statement and its related settings, it automatically creates detector instances from the image the user chooses and builds an auto-scaling group to accommodate the instances. Within each instance, Heat further invokes a script we built to configure the IDS engine with a set of default parameters. For example, `NumFileBolts` defines how many bolts the user wants Storm Engine to start for processing file-related system calls. The user can also change the default value of those parameters when submitting the template for stack creation through Heat API. We further developed configuration scripts to run inside each instance generated by the user's application to

set up an auditing daemon that streams out the instance's audit trails during its runtime.

Stream-Based IDS Platform. At the receiving end of the audit-trail streams are the detector instances. In our research, those instances were all built on top of the Storm stream-processing engine. As described before, a Storm cluster includes two types of worker nodes, sprouts and bolts, which connect to each other to form a network structure as part of the system's configuration. For simplicity, our implementation just utilizes one sprout node to receive streams from application instances and dispatch them to different bolt nodes. The latter can be added at the system's runtime to handle extra workload brought in by the application during its auto-scaling process. When this happens, each new bolt is dynamically connected to the sprout. A direct use of this stream-based IDS platform is just to let the stream from each application instance be taken care of by one bolt. For this purpose, the bolt was built to gather context information from a database maintained by Heat and upload data such as intrusion signatures or normal behavior patterns of the application from Storm's internal database to its memory, which we elaborate later. With such supports, a classic signature-based or anomaly IDS can directly run within the bolt to process the audit trails from application instances.

A problem for this design is the lack of cooperations among bolts. Such cooperations can potentially improve the performance and flexibility of the whole detection service. In our research, we implemented a simple Systrace-like system-call inspection system, which is designed to concurrently process audit trails from hundreds of application instances, each containing a large number of calls. Those calls are checked one by one against the behavior profile of each instance, including the names and parameter patterns of the calls considered to be normal for the instance. Those profiles are maintained within the detector database. A small set of them are created offline when the whole template stack (the cloud application and the detection service) runs in a training mode. Most of them, however, are generated online through profile reuse within an auto-scaling group or across different groups, which we describe in Sect. 3.3.

The operations of an application often generate a huge amount of auditing data. For example, opening Firefox and using it to perform a Google search can produce as many as 36936 system calls. To avoid the performance overheads on both the application instance and the detector, our system-call inspector only focuses on several categories of calls considered to be inevitable for an intrusion to succeed. More specifically, what are currently inspected in our detector include those for operating on file systems (e.g., `open`), networking (e.g., `connect`) and generating a remote shell (e.g., `execve`)[2]. For the Firefox example, this means the inspector needs to check only 44 (instead of 36936) system calls. We further designed the detection service in a way that a group of bolts were tuned to processing a single category of calls only. This allows the bolts to work more

[2] Those calls need to happen on almost all intrusion vectors (as evidenced by our false negative evaluation in Sect. 4.2). Also our design can be easily extended to accommodate other types of calls.

efficiently on the data, helps better balance the workload across different bolts and most importantly makes it possible to integrate other detection mechanisms into the system. Specifically, bolts working on the same category only need to maintain the profiles of the system calls in that category across all auto-scaling groups, instead of complete profiles of those groups in the case that one bolt is assigned with the whole stream from a random application instance within the template stack. Also, depending on the number of calls observed in each category, the system can dynamically increase or decrease the number of bolts associated with the category to better allocate the resources of the detection service. Further, this treatment automatically organizes the outputs of our detector into categories: all file operations are assigned to one set of bolts and network activities are given to the other set, etc. As a result, we can conveniently let those bolts stream their outputs to another set of bolts that run other detection mechanisms. Particularly, in our implementation, we added a group of bolts that run Snort on the content produced by network calls, which are transmitted from the workers associated with the network category.

为了提高效率，让一个bolt处理一类系统调用

Our implementation serving this purpose is illustrated in Fig. 3. We built two layers of bolts. On the first layer are *dispatchers* that parse streams, extract system calls, group them into vectors with various lengths depending on the category of the calls. A simplified form of such a vector is (`mac`, `ID`, `program`, `name`, `parameters`), which describes the MAC address of the instance from which the call was initiated, its identifier within the cloud (including the number of its stack and the name of its auto-scaling group), the program that made the call, the name of the system function called and its parameters. Note that the identifier `ID` is left blank by the dispatchers, which cannot directly observe them. Based upon `name` (the call type it is associated with), the vectors are regrouped. Those within the same category (file operations, networking, etc.) are streamed to the same group of worker bolts on the second layer. The worker bolts maintain the profiles of different instances within their memory, which only include the functions in the category and their parameters. During its runtime, each worker uses a vector's `mac` to locate the profile for a specific instance and checks whether other elements of the vector, such as `program`, `name` and `parameters`, are in compliance with it. These workers also retrieve the identifier for each MAC address from the orchestrator, which is used for profile sharing. Behind those bolts, our implementation can accommodate other layers of bolts in the user's request, for the purpose of signature-based detection and attack graph reconstruction.

Automated Orchestration and Scaling. Although the detection service is configured by Heat and automatically scales through Heat, the communication between them is quite limited during runtime. Particularly, the service does not know what happens in the user's cloud application, for example, when a new instance is created and which auto-scaling group the instance belongs to. Without such information, it is almost impossible for the service to properly configure new detector instances in response to the dynamics within the application. To address this issue, we built into Heat a mechanism to facilitate its communication

探测器除了需要instance的系统调用信息，还需要heat提供有关instance的上下文信息。如新instance创建时间，所属的group。

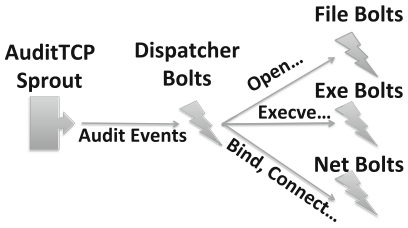

Fig. 3. Detector Storm topology

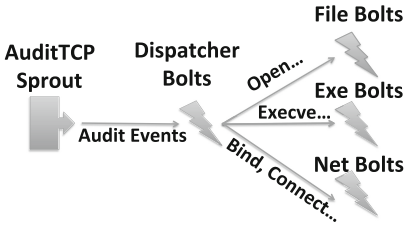

Fig. 4. Automated Orchestration and Scaling

with the detection service at runtime, through its OpenStack Heat database and Storm's internal database. Figure 4 illustrates how the mechanism works.

Specifically, we modified Heat's database, adding in three tables (`ids_stack`, `ids_autoscaling` and `ids_instance`) for a user's stack, auto-scaling groups and application instances respectively. Among them, `ids_stack` is for profile reuse across different stacks, `ids_autoscaling` maintains all the auto-scaling groups within the same stack and `ids_instance` keeps all instances within the same group. Those tables are utilized by Heat when building up the whole stack and dynamically adjusting its infrastructure, adding or removing instances from different auto-scaling groups. In this way, the orchestrator keeps record of the provenance for each instance, i.e., the stack and auto-scaling group it belongs to. Such information is saved to the `role` attribute of the instance. In our implementation, we instrumented Heat engine functions `EngineService.create_stack`, `resources.AutoScalingGroup._handle_create`, `StackResource.create_with_template` and `resources.instance.check_create_complete` to operate on those tables and track the provenance of each instance.

在heat中添加了3个table来维护对用户软件栈，组和instance的信息。

出处，起源

Through the instance table, the Heat orchestrator can mark the status of a stack as "training" or "enforcement". This mark is then passed to the detector that queries the Heat database for instance information. Such a query needs to go through proper authentication, which in our research is based upon the cloud user's credential, as the database is shared among different users. With the state information from the orchestrator, the detection service will decide whether to learn a behavior profile (which is kept within Storm's internal database) of an instance or go ahead to detect its suspicious behaviors using the existing profile or malicious activities by looking for known signatures. When the application adds in new instances within its auto-scaling group, the detection service will notice that new MAC addresses show up, whose profiles and other information are not present in the internal database. In this case, the service will query the instance table in the Heat database for a newly observed MAC address to get the `role` of its instance. This attribute, as elaborated before, contains the provenance of the instance, which enables the detector to figure out how to reuse existing profiles for protecting the instance in an anomaly detection.

3.3 Context-Aware Profile Configuration

Challenges in Anomaly Detection on the Cloud. As mentioned before, the high-performance IDS platform within Elite can support different kinds of detection techniques. For signature-based detection, all we need to do is just running the existing mechanism within the bolts tasked to process the whole streams of individual application instances. When it comes to anomaly detection, however, we have to consider the complexity introduced by profile learning. Specifically, there should be an off-line learning stage during which training traffic is used to drive the operations of the user's application. The audit trails produced thereby are analyzed by the detection service for constructing different instances' profiles. More challenging here is profiling a newly created instance during the application's runtime, which needs to happen in real-time. Although it is conceivable that the new instance will behave in a similar way as others within the same auto-scaling group, subtle differences can still exist in their profiles. The problem we faced in our research is how to automatically construct an accurate profile without going through the learning stage, both for the instances within an existing auto-scaling group and for those in a newly created stack. Following we elaborate a set of techniques that facilitate reuse of profiles within one group and cross users.

想得到的
微小的
profiles的重用

Profile Generation and Reuse. Profile learning is fully supported by Elite. Once a user selects an anomaly detection image when building her template, the Elite components within Heat automatically set the application's execution mode to "training" as soon as the stack is deployed. Through Heat, our implementation creates a set of instances, based upon the user's specification, to run scripts[3] that generate training traffic for the whole stack. For example, these instances can produce HTTP traffic to a web application running on top of Apache servers deployed in application instances. In the meantime, the detection service learns from individual instances' audit trails invariants in their behaviors and save such profiles to the detector's database. Also, all the profiles from different instances within the same auto-scaling group are generalized into a profile template. During the system's runtime, whenever a new instance is created for a group, its template signature is then specialized according to the unique feature of the new instance to provide it immediate protection.

在detector刚被部署时，首先会进入trainning模式，通过训练来生成不同group的profile模版。

For the Storm-based system call inspector implemented in our research, its learning phase involves 2 to 5 instances per auto-scaling group. Each of these instances is monitored by a set of concurrently running detector instances. As discussed before, those detectors extract from the application instance's audit trail system-call vectors and classify them into different categories according to the types of the calls (e.g., all those related to file operations). The recipients of the vector stream within one category, the worker bolts, further group all the vectors using their MAC addresses, program names, specific system call names and others, and removes duplicated ones. For example, all the system calls `open`

[3] An example here is JMeter Script Recorder, which can be provided by the cloud and customized by the user.

from a given `mac` and a specific program are placed inside one group and for each vector within the group, others with the exactly same parameters are dropped. The worker bolts further attempt to generalize call parameters across the vectors within the group. Particularly, for each outgoing network call, they contact the Heat orchestrator to find out whether the call is made to another auto-scaling group: for example, a web server instance accesses an instance within the auto-scaling group of database servers. Note that such connections are typical for a cloud application, which actually describe the topology of its whole stack. What our approach does here is to replace the IP address within the parameters for such a network call (e.g., `connect`) with the identifier (e.g. `102-DBServerGroup`, where `102` is the number of the stack and the rest part is the group name) of the target auto-scaling group. This step is necessary for reducing the false positives of the profile generated by the bolts, which comprises all those generalized vectors for a specific MAC address and is stored under the identifer of the application instance or its auto-scaling group.

The detection service further compares the profiles from multiple instances within the same auto-scaling group to generate a profile template. Specifically, the vectors that appear across all profiles are directly moved to the template. For other vectors, our approach inspects them one by one, across the profiles, looking for the invariants in the parameters of the same system call (from the same program) and the strings that match a set of predetermined patterns (e.g., the instance's identifier). For example, consider the call `open(/var/lib/cloud/i-0000010a/config)` in one profile and `open (/var/lib/cloud/i-0000010b/config)` in another. The vector (`mac`, `ID`, `program`, `open`, `/var/lib/cloud/(instance_id)/config)` will be added to the template. Note that `mac` is left blank here, which needs to be filled with the MAC address of a new instance and the content "`(instance_id)`" matches the instance's ID and is therefore annotated for the specialization purpose.

During the system's runtime, whenever a new instance within the same auto-scaling group is created, the detection service specializes the profile template to generate one for the instance. The idea here is to replace wildcards with the concrete value observed from the new instance's operation, once the call name, related parameter invariants and other elements (e.g., `program`) are matched. In the above example, as soon as the service finds that the new instance makes a call `open(/var/lib/cloud/i-0000010c/config)`, the aforementioned vector is immediately specialized into (`mac`, `ID`, `program`, `open`, `/var/lib/cloud/i-0000010c/config`) and added into the instance's profile. Note that in the case that no invariant can be found in the parameters of the same calls across all instances, the whole parameter part of the vector within a profile template is replaced with a wildcard "`*`". This notifies the detection service that for a new instance, if it makes the call within the vector, any parameter of the call will be acceptable.

Reuse Across Users. The above profile-sharing technique makes it possible to run anomaly detection on an auto-scaling cloud application. What is also desired here is to shorten or even completely remove the learning phase that

bootstraps the detection mechanism. To this end, we investigated the technique that supports profile reuse across auto-scaling groups, even across different cloud users.

A key observation is that whenever two application instances run an identical software stack (e.g., OS/web server/web application) with identical configurations, their behavior profiles should be very similar. Indeed, in Sect. 4.2, we present our study on popular stacks, which shows that they produced the same set of system calls with very similar parameters. Further, even in the presence of small discrepancies in the configurations, as long as critical components remain unchanged (such as plug-in settings for `Joomla!`), the profiles from those instances often still come close to each other. In these cases, we can reuse the profile template from one auto-scaling group on the other one to avoid the off-line learning stage[4], and instead adjust the template and the profiles derived from it during the new group's runtime whenever false positives show up. Also, given that people tend to make minor customizations on popular software stacks with default settings, there are lot of chances to reuse profiles even across different cloud users.

What stands in the way of such a reuse, however, is privacy concerns. Specifically, cloud users may not be willing to expose all her software settings to the cloud, which may reveal potential security weaknesses in her system [20]. In this case, the template file one submits to the orchestrator may only describe part of her software stack and some configurations can happen within each application instance using the scripts provided by the user. Note that even though the cloud service provider can figure out such information by inspecting the content of the user's VM instances, they are reluctant to do so and afraid of legal liabilities. In our research, we designed a simple mechanism that facilitates the reuse of profiles across users without leaking out configuration information to either the cloud or the parties who adopt different settings, and also the identities of the parties involved.

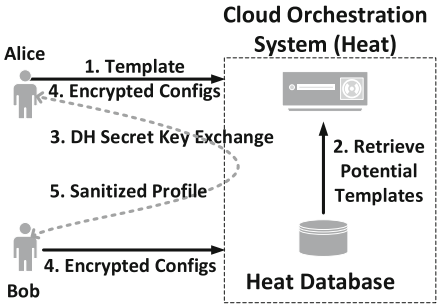

Fig. 5. Privacy-preserving profile-sharing

[4] False positives incurred by such profile sharing can be further adjusted during the system's online operation.

Figure 5 illustrates the way the mechanism works. As soon as a user submits her template file to the orchestrator to deploy a stack with the detection service, the Elite component within Heat searches the table `ids_stack` for other template files that contain auto-scaling groups or instances with identical software stacks and configurations. For each of such template files discovered, which may not document the full configurations made by its owner, the cloud contacts the owner to further compare her configurations with those of the new user in a privacy-preserving way. Specifically, both users, without knowing each other, exchange a secret key through the cloud using the classic Diffie-Hellman (DH) key exchange protocol. The DH protocol is designed to establish a secret between strangers over an insecure channel, as long as the party eavesdropping on the channel (the cloud) is considered to be honest but curious, never inserting its own messages into the channel to play a man-in-the-middle. Using the exchanged secret, these two users can compute keyed hash values for the value part of each key-value pair within their system configuration files for an instance and submit them to the cloud for a comparison. If the cloud finds that their configurations are identical or very similar, one party can anonymously share her profile template for the instance to the other through the cloud. Specifically, the party first searches for the occurrences of instance IDs within the profiles, and removes them but sets indicators there to let the recipient fill in his own IDs. Then, she continues to sanitize other content of her profile template, replacing confidential information with the indicators for the types of data that should be in place. This template is then encrypted using the shared secret key and delivered to the recipient through the cloud. This way, profiles are reused anonymously, without disclosing sensitive information to the cloud and the party with different settings.

For example, Alice joins a community organized by the cloud, in which every user is committed to sharing her profiles to others when needed in a privacy-preserving way, and also benefits from other profiles in accelerating her deployment of cloud applications. In this case, Alice wants to create a website using a WordPress CloudFormation template [21]. After the template is submitted, the cloud orchestrator first compares it with other templates in its database: if another user, say Bob, has utilized a very similar template file before, Alice might be able to reuse his profile. To further assess the possibility of reuse, Alice and Bob need to compare the configurations for their software such as Apache HTTP server and WordPress. Because their configurations may contain sensitive data like passwords (WordPress involving MySQL password), Alice and Bob cannot do this in plaintext. Instead, they exchange a secret key K using the DH protocol through the cloud, and then encrypt their software configurations. Specifically, for each key-value pair within their configuration files, the value part is encrypted using K. The ciphertext here is sent to the cloud, which compares them to find out how similar these two configurations are. If they are identical except for some minor keys (e.g., WordPress Database Table prefix), Alice and Bob are instructed by the cloud that the profile can be reused. Then Bob sanitizes his profile automatically by removing the content of private items, including secrets such as passwords, unique identifiers like instance IDs, host names and

IP addresses, and other information like installation path, but annotates each item with its content type (e.g., password, host name, etc.). The profile is then encrypted under K and handed over to Alice through the cloud. The recipient here, Alice, fills in the blanks (sanitized items) with her information before running the profile to protect her website.

As we can see from the example, at the end of this procedure, Alice and Bob do not know each others' identities. They do not have exact information about the overlap of their configuration files, not to mention the parts that differ from each other. The cloud knows the identity of both parties but has no idea about the values of their configuration settings (other than how similar the two configurations are).

4 Evaluation

To understand how Elite performs in practice, we tested our prototype against real-world security threats and heavy computing tasks. What we want to find out includes the impacts of our profile reuse approach on the effectiveness of the detectors running on top of Elite, and the performance of the framework in protecting the cloud application with a dynamic,intensive workload. In this section, we report the results of the study.

4.1 Settings

Our evaluation study was conducted under the following system settings:

The Cloud and Orchestrator. The cloud used in our study includes 22 workstations, each equipped with a 4-core 3.10 GHz Intel i5-2400, 8 GB memory and an 80 GB local disk. On those workstations, we deployed an OpenStack (Icehouse) cluster with 1 controller node and 21 compute nodes. The Heat orchestration service within OpenStack was modified to accommodate the Elite components we implemented. The instance used in our experiments was typically configured with 1 core, 2 GB memory and 10 GB storage and ran Fedora 17 with heat-cfntools [22] to support cloud orchestration.

Cloud Applications. Multiple applications were run on top of this cloud infrastructure to evaluate the effectiveness and performance of our techniques. Specifically, three popular content-management systems, `WordPress`, `Drupal` and `Joomla!`, were used to understand the effectiveness of profile reuse (see Table 1). Also serving this purpose were two prominent penetration testing platforms: `Kali Linux` and `Metasploitable2`. WordPress was further utilized in our performance evaluation.

The Detection Service. The detection service was built within a Storm 0.9.1 engine, with a default configuration, in which a single sprout node was connected to multiple bolts. Each detector node was hosted within a typical VM instance (1 core, 2 GB memory and 10 GB disk). The whole service was set to be able to automatically scale in the presence of dynamic workloads.

Table 1. Workloads for false positive evaluation.

Stack Application	Automatic (1000 Users)	Manual
WordPress	Create User, User Login, Browse Blog, Post/Comment Blog, Reply Comment	Change Theme, Activate Widgets, Change User Role, Add/Edit Media in Library
Joomla!	User Registration, User Login, Create/Browse/Edit Article	Enable/Activate User, Add Menu, Enable/Disable Plug-ins, Edit/Publish/Unpublish Modules
Drupal	User Registration, User Login, Browse/Add Article, Add Comments	Unblock User, Install/Enable/Disable Theme, Add Role, View Reports, Enable/Disable Module, Change Site Config/Structure

4.2 Effectiveness

For anomaly detection in general and our prototype service in particular, the most important issue we want to understand is the effectiveness of profile reuse through Elite. To this end, we measured in our study the false positive and negative rates of the profile derived from a profile template, as described in Sect. 3.3, under different web traffic and real-world exploits. The results show that our approach does not undermine the accuracy of a detection system, and instead, makes it convenient to use in a cloud environment through swiftly deploying profiles for new instances acquired by a cloud application.

False Positive. We ran `WordPress`, `Drupal` and `Joomla!` on Apache 2.2.23 under the Elite-enhanced orchestrator to find out whether the new profile automatically reused causes more false alarms than the one constructed through profile learning. In the experiments, all those web services were configured using sample templates provided by the AWS CloudFormation website [23].

Once deployed under Elite, the cloud stack running these applications first operated in learning mode. To generate a workload as realistic as possible for the study, we leveraged one of the most wildly used load testing tool, JMeter, to simulate 1000 users who produced random requests to explore common functionalities of those web applications (e.g., browsing blogs). To complement those requests, we further performed administrative operations manually on a set of application instances. The complete list of the activities can be found in Table 1.

The learning stage ended up creating profile templates for the auto-scaling groups hosting those applications, which are used to generate new profiles for new instances added to the groups. Direct testing on the new instances is complicated, since the traffic is automatically distributed to all the instances within an auto-scaling group, including existing ones. What we did in our research is to create a new stack using the same template file for the old one and apply the profile templates (from the old cloud orchestration stack) to their corresponding groups

Table 2. False Positive Results

Stack Application	Profile			Auditing Events	False Positives
	File	Exe	Net		
WordPress	4313	245	45	~271120	11
Joomla!	7427	306	41	~577040	8
Drupal	4945	334	40	~190400	9

on the new stack, which were specialized automatically during the operation of the new stack[5]. This new stack was directly set to the enforcement mode, in the presence of requests (which were different from those used in learning stage) from JMeter, for a false-positive measurement.

Table 2 shows the experiment results. The false positive rates for all these three web applications were found to be exceedingly low, around 10 over hundreds of thousands of auditing events. Most importantly, comparing the profile learnt (on the old stack) and the one derived (on the new stack), the false positives observed are identical: all caused by network-related system calls. For example, one false alarm from WordPress came from the connection of `httpd` to a different IP address than the one observed during the learning phase, which all belong to `WordPress.org`.

Interestingly, we found that all the MYSQL database server instances across the stacks for `WordPress`, `Drupal` and `Joomla!` had very similar profiles, allowing the profile template from one stack to be automatically specialized to protect the instance in the corresponding auto-scaling group within another stack. This also happened to the load balancers across these stacks. All these instances (database servers or load balancers) were installed with identical software stacks and configured in the same way across the stacks. However, they are working on completely different types of data, serving different web applications. Our findings show that it is realistic to share profiles between the instances within different stacks and belonging to different users, as long as they all have the same software stacks and configurations.

Note that such profile sharing cannot be achieved by directly applying a profile learnt from one instance to another, even when both instances belong to the same auto-scaling group. We actually found the presence of significant differences within some vectors in different instances' profiles, which were automatically identified and specialized. An example is the `open` call made to a random file under the directory `/var/lib/mysql/`, as illustrated in Fig. 6. This file name is within the profile learnt but varies across different instances. Our approach automatically identified its invariant patterns (highlighted in Fig. 6), which were used to specialize the profile. Should such a profile be directly reused without specialization, much more false alarms would be produced.

[5] In addition to the contents with wildcards, those profile templates were also specialized according to the ID of the stack.

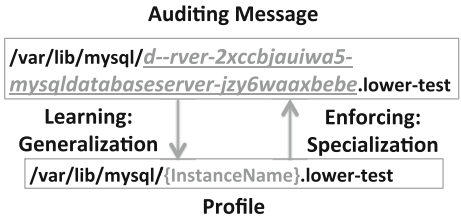

Fig. 6. An example of profile generalization/specialization.

False Negative. To study whether profile reuse could cause the detector to miss the attacks it should be able to catch, we utilized two well-known penetration testing platforms: `Kali Linux` and `Metasploitable2`. Kali Linux is a Linux distribution built for advanced and versatile penetration testing. It integrates more than 300 tools, including the `Metasploit framework`, a tool for developing and executing security exploits. `Metasploitable2` is an intentionally vulnerable Linux VM image (based on Ubuntu 8). It contains a collection of outdated vulnerable and improperly configured software and services for testing security tools.

In our experiments, we deployed `Metasploitable2` over a cloud stack and again, ran the stack first in learning mode. During this process, a script was used to generate requests, causing the application instance (hosting Metasploitable2) to perform different operations. For example, for Samba (SMB) and FTP, the script made connections to their service daemons, listed files, and uploaded/downloaded a set of files. For Apache 2, the script ran two crawlers (a python crawler and a `wget` based crawler) to crawl its web content. To gener-

Table 3. False Negative Results.('rev' is a shortcut for 'reverse')

Exploit	Payload	Detection	
		Baseline	Elite
Samba usermap script	cmd/unix/reverse	Yes	Yes
	cmd/unix/rev_netcat	Yes	Yes
Samba Symlink Traversal	-	No	No
vsftpd 2.3.4 Backdoor	-	Yes	Yes
UnrealIRCd 3.2.8.1 Backdoor	cmd/unix/reverse	Yes	Yes
PHP-CGI Arg Injection	generic/shell_rev_tcp	Yes	Yes
dRuby Code Exec	cmd/unix/reverse	Yes	Yes
	cmd/unix/rev_netcat	Yes	Yes
Java RMI Server Code Exec	java/shell/rev_tcp	Yes	Yes
	linux/x86/shell_rev_tcp	Yes	Yes
DistCCd Command Exec	cmd/unix/reverse	Yes	Yes
Detection Rates		11/12	11/12

Table 4. Metasploitable2 Selected Vulnerabilities.

Target	Description	CVE / OSVDB
Samba	Usermap script – Command Injection	CVE-2007-2447
Samba	Symlink Directory Traversal	OSVDB-62145
vsftpd 2.3.4	Backdoor – Command Execution	CVE-2011-2523
UnrealIRCd 3.2.8.1	Backdoor – Command Execution	CVE-2010-2075
PHP-CGI	Argument Injection	CVE-2012-1823
dRuby [24]	DRB Remote Code Execution	-
Java RMI Server [25]	Java Remote Code Execution	-
DistCCd	Command Execution	CVE-2004-2687

ate a realistic workload for `DistCCd`, a distributed compilation tool, the script requested a Metasploitable2 instance to compile the source code of `sqlite3` [26].

The profile templates created in learning stage were then specialized for the instances within a new Heat stack generated from the same orchestration template file. Again, this new stack operated in enforcement mode and ran `Metasploitable2` in the presence of exploit attempts made from Kali. Table 4 lists 12 exploits tested in our study, including 8 attacks with different payloads, which will cause a compromised system to behave differently (e.g., spawning a shell). All of them led to successful attacks. Using the original profile ("Baseline" in Table 3), which is the one built up during the learning stage, the detector caught 11 of these exploits, all except Samba symlink traversal. After replacing the original profile with the derived one (specialized from a shared profile template), we observed that Elite detected the exactly same set of exploits (11 out of 12). Note that the Samba symlink traversal exploit [27,28], which provides access to the victim's file system, was missed in both cases, due to the incomplete set of system calls monitored in our prototype. We emphasize that this problem is caused by the underlying detection mechanism, not by the reuse of profiles. The results actually strongly indicate that our profile-reuse approach will not affect the accuracy of a detector.

4.3 Performance

We further studied the performance of the Elite-enhanced orchestration in terms of its consumption of computing resources and its impacts on the cloud user's experience. Specifically, we measured the overheads incurred by our implementation when collecting audit trails from individual application instances and streaming them out to the detection service. Then we evaluated how our elastic detectors help control the time for processing audit trails and the amount of resources required for this purpose.

Overheads. The only overheads brought in by Elite to individual application instances come from 3 audit-related processes for generating, dispatching and streaming out auditing events. These processes are `auditd`, `audispd` and

Table 5. Overheads of Audit Processes.

Process	Peak Memory (KB)			Peak %CPU
	Virtual (VSZ)	Physical (RSS)	%MEM	
auditd	91768	524	0.03	1
audispd	80692	556	0.03	1
audisp-remote	6876	492	0.02	0.33

`audisp-remote`. To measure this cost, we set up an instance with WordPress installed and ran JMeter to simulate 100 concurrent users, automatically generating workloads as described in Table 1. We can see that the CPU and memory usages of the processes are very low from Table 5.

The communication cost for running Elite was mainly caused by streaming out the audit trails to detectors. For example, on one WordPress instance, during the process of installing WordPress and handling 100 concurrent users' requests for around 30 min, audit dispatcher needs to stream out ˜10 MB of auditing events. Given the average bandwidth between instances in our OpenStack setup is ˜2.5 MB/s and bandwidths provided by public clouds are even higher [29], so this level of bandwidth consumption is rather low for a cloud application and does not affect its normal operation. For the Heat orchestrator, the performance impact of our approach is unobservable, due to a large workload it already undertakes to build up the whole stack and coordinate its operations.

Elastic Detection. To understand the important support Elite provides to the intrusion detection on the cloud, we compared the performance of detection with and without the elastic service offered by Elite. Here we measured the performance in terms of the average delay in processing an auditing event (called *Average Message Complete Latency* or AMCL). This latency describes the average duration from the moment a detector receives an auditing event to the time when this event is fully analyzed. To get AMCL, we set up two stacks using the standard WordPress CloudFormation template (with and without Elite) and gradually increased their runtime workloads. For the stack with elastic detection, we utilized a simple yet conservative policy that set the auto-scaling ratio to 1:5, e.g., there were 12 detector instances when the number of application instances grew to 60. Note that our design can also support other auto-scaling policies based on CloudWatch's alarm mechanism, for example those based upon CPU usages. As we can see from Fig. 7, in the absence of the elastic detection service, the AMCL became prohibitively large when the number of application instances went up to 30 (which were all served by a single detector instance). This basically means that system administrator can only find out an attack event more than 10 min after it actually happens on an application instance. We tried twice to scale the stack to 60 application instances without elastic detection, unfortunately Storm IDS engine inside the single detector crushed simply because it just can't handle events from so many application instances. By comparison, Elite automatically added in more resources for detection when large workloads

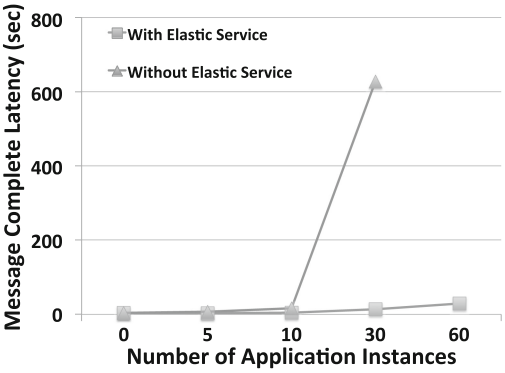

Fig. 7. Average message complete latency with and without elite.

Table 6. Performance of privacy-preserving configuration matching. The elapsed time is reported in seconds.

DH Key Exchange Derivation		Key-Value Pairs Matching	
DH Generation	DH Shared Key	82 Pairs	1000 Pairs
0.05	1.36	0.41	1.46

came in, which results in a relatively stable AMCL: even in the presence of 60 application instances, the latency observed in our study is still below 30 s.

Reuse Across Users. We built a prototype implementation of our privacy-preserving profile sharing subsystem with the goal of evaluating the performance of two key components: DH key exchange and privacy-preserving configuration matching. The results are shown in Table 6. As we can see from the table, the whole operation is very efficient, which is always done within 2 s.

5 Discussion

Our evaluation study shows that Elite works effectively against dynamic and intensive workloads from the popular applications it protects, without undermining the accuracy of detection. On the other hand, our current design and implementation are still preliminary. Much needs to be done to further improve its capabilities, which we discuss as follows.

Platform. Our design extends the cloud-formation language to allow the user to request a detection service to be set up according to a set of parameters. What can be done next is to further enrich the language for configuration of a complicated security service, including not only a single type of intrusion detectors and their combination but also other protection mechanisms, like integrity checker and a forensic analysis mechanism. Also, to support those new security

components, changes need to be made to the orchestration engine to provide the components stack-wide information, particularly interactions between different instances.

Protection Service. The prototype we implemented just includes a parallel detection service. Follow-up research could build other aforementioned protection mechanisms on top of the stream processing engine. Also in addition to serving individual users, the security service offered by the cloud can leverage its global view to help detect malicious code or activities and prevent them from propagating to other cloud users. Such a service becomes increasingly important under the current trend in moving organizational infrastructures to commercial clouds.

Profile Reuse. As a very preliminary step toward leveraging stack-wide or even cloud-wide information for security protection, we developed a suite of techniques for sharing normal behavior profiles across auto-scaling groups and different users. However, the effectiveness of those techniques need to be further evaluated against different types of detection systems. Also likely the design itself needs to be adjusted to make profile-reuse work on those systems.

Instance Migration. Due to regular infrastructure maintenance, cloud providers may need to migrate instances across physical machines. A concern is that such migration could interrupt the executions of detector and application instances. However, with new technologies like live migration [30] being adopted by cloud providers today, such an infrastructure maintenance can be transparent to instances and therefore will not affect Elite's normal operations.

6 Related Work

Traditional Intrusion Detection. Intrusion detection has been studied for decades. Numerous approaches have been proposed for signature-based and anomaly detection. Techniques are developed for host-based [31,32] and network-based IDS [33], using various machine learning [34] and data mining [35,36] algorithms. Moving those conventional techniques to the cloud faces great challenges, due to the dynamics in the cloud environment. Elite is designed to address those challenges, offering a high-performance platform to support those detection techniques on the cloud.

Cloud-Based Intrusion Detection and Prevention. Cloud-based IDS can leverage the information collected outside VM instances. For example, detection systems [37,38] were built to run on the hypervisor to support no-intrusive inspections of events that happen inside VM instances. Such detectors can also be accommodated by the Elite platform, which can operate a centralized service to analyze all the events of individual instances gathered at hypervisor level.

Most related to our work is CIDS [39], which constructs a peer-to-peer (P2P) network on top of application instances with embedded detectors. Through this P2P infrastructure, detectors collaborate with each other, sharing resources and

balancing their workloads. Also, a few conceptual designs of centralized detectors have been reviewed in [40,41]. Virtual middleboxes framework like Stratos [42] has also been proposed for hosting IDS in an elastic way. However, to our knowledge, no effort has ever been made to integrate a centralized detection service into the cloud orchestration mechanism, which is crucial for protecting a highly dynamic, auto-scaling computing system.

System Call Based Host Protection. The anomaly detector implemented in our prototype is a parallelized version of system-call-policy based protection. Examples for this type of detection techniques include Systrace [15], BlueBox [43], SELinux [44], AppArmor [45] and TOMOYO Linux [46]. An extension of these approaches, which largely enforce security policies at individual calls, is call-sequence based anomaly detection [12,47]. Such techniques can also be conveniently supported by our elastic detection platform to protect cloud applications in a large scale.

7 Conclusion

The dynamics of the cloud-computing environment makes it extremely challenging to protect a cloud user's application. A detection service designed for the cloud is expected to continuously monitor the state of the application, dynamically adapt its computing scale to workloads and reallocate its existing resources and configure new resources at runtime. This requirement can only be met through close interactions with cloud orchestrators. In this paper, we describe Elite, the first elastic detection platform designed for this purpose. Elite is developed to enhance existing cloud orchestrators, enabling the user to conveniently request a detection service and specify its parameters through her cloud-formation template. Such a detection service is built upon a high-performance stream-processing engine, capable of concurrently analyzing a large number of audit streams and automatically adjusting its computing scale. The service is further supported by the Elite components within the orchestrator, which timely updates the application's state information. To avoid learning a new instance's behavior profile for anomaly detection, which is unrealistic during the application's runtime, we studied a set of new techniques to facilitate profile reuse within an auto-scaling group, and also across users in a privacy-preserving way. Our evaluation shows that such profile sharing does not undermine the accuracy of detection, and also the whole system effectively handles heavy workloads produced by popular web applications.

Acknowledgments. The project is supported in part by National Science Foundation CNS-1117106, 1223477, 1223495, 1223967, 1330491, and 1408944. Yangyi Chen was also supported in part by IBM internship program. The views and conclusions contained herein are those of the authors only and do not necessarily reflect those of the NSF or IBM.

References

1. Somorovsky, J., Heiderich, M., Jensen, M., Schwenk, J., Gruschka, N., Lo Iacono, L.: All your clouds are belong to us: Security analysis of cloud management interfaces. In: CCSW (2011)
2. Mulazzani, M., Schrittwieser, S., Leithner, M., Huber, M., Weippl, E.: Dark clouds on the horizon: using cloud storage as attack vector and online slack space. In: USENIX Security (2011)
3. McAfee SaaS Endpoint Protection Suite. http://www.mcafee.com/us/products/saas-endpoint-protection-suite.aspx
4. Trend Micro Deep Security as a Service. http://www.trendmicro.com/us/business/saas/deep-security-as-a-service/index.html
5. Alerg Logic Public Cloud Security. https://www.alertlogic.com/products-services/public-cloud-security/
6. Heat - OpenStack. https://wiki.openstack.org/wiki/Heat
7. AWS CloudFormation. https://aws.amazon.com/cloudformation/
8. Sung, A.H., Xu, J., Chavez, P., Mukkamala, S.: Static analyzer of vicious executables (save). In: ACSAC, Washington, DC, USA (2004)
9. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.A.: Behavior-based spyware detection. In: USENIX Security, Berkeley, CA, USA (2006)
10. Kolbitsch, C., Comparetti, P.M., Kruegel, C., Kirda, E., Zhou, X., Wang, X.: Effective and efficient malware detection at the end host. In: USENIX Security (2009)
11. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis. In: CCS, New York, USA (2007)
12. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. J. Comput. Secur. **6**, 151–180 (1998)
13. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for unix processes. In: IEEE S&P (1996)
14. Michael, C.C., Ghosh, A.: Simple, state-based approaches to program-based anomaly detection. ACM Trans. Inf. Syst. Secur. **5**, 203–237 (2002). http://doi.acm.org/10.1145/545186.545187
15. Provos, N.: Improving host security with system call policies. In: USENIX Security (2002)
16. IBM InfoSphere Streams. http://www-03.ibm.com/software/products/en/infosphere-streams
17. Storm - The Apache Software Foundation! http://storm.incubator.apache.org/
18. Apache Storm - A system for processing streaming data in real time. http://hortonworks.com/hadoop/storm/
19. Apache ZooKeeper. http://zookeeper.apache.org/
20. Google Hacking Database. http://www.exploit-db.com/google-dorks/
21. AWS CloudFormation Sample Template WordPressMultiAZ. https://s3-us-west-2.amazonaws.com/cloudformation-templates-us-west-2/WordPress_Multi_AZ.template
22. Heat API Instance Tools. https://launchpad.net/heat-cfntools
23. AWS CloudFormation Templates. https://aws.amazon.com/cloudformation/aws-cloudformation-templates/
24. Distributed Ruby Send instance eval/syscall Code Execution. https://www.rapid7.com/db/modules/exploit/linux/misc/drb_remote_codeexec

25. Java RMI Server Insecure Default Configuration Java Code Execution. https://www.rapid7.com/db/modules/exploit/multi/misc/java_rmi_server
26. SQLite Home Page. http://www.sqlite.org/
27. Samba Guest Account Symlink Traversal Arbitrary File Access. http://www.osvdb.org/62145
28. Samba Symlink Directory Traversal. https://www.rapid7.com/db/modules/auxiliary/admin/smb/samba_symlink_traversal
29. Need for speed: Testing the networking performance of the top 4 cloud providers. http://gigaom.com/2014/04/12/need-for-speed-testing-the-networking-performance-of-the-top-4-cloud-providers/
30. Google Compute Engine: Transparent maintenance. https://developers.google.com/compute/docs/zones#maintenance
31. Kim, G.H., Spafford, E.H.: The design and implementation of tripwire: a file system integrity checker. In: CCS, New York, USA (1994)
32. Vigna, G., Kruegel, C.: Host-based intrusion detection (2005)
33. Roesch, M.: Snort - lightweight intrusion detection for networks. In: USENIX System Administration, Berkeley, CA, USA (1999)
34. Tsai, C.-F., Hsu, Y.-F., Lin, C.-Y., Lin, W.-Y.: Intrusion detection by machine learning: a review. Expert Syst. Appl. **36**, 11994–12000 (2009)
35. Lee, W., Stolfo, S.J., Mok, K.W.: A data mining framework for building intrusion detection models. In: S&P (1999)
36. Lee, W., Stolfo, S.J., Mok, K.W.: Adaptive intrusion detection: a data mining approach. Artif. Intell. Rev. **14**, 533–567 (2000)
37. Azmandian, F., Moffie, M., Alshawabkeh, M., Dy, J., Aslam, J., Kaeli, D.: Virtual machine monitor-based lightweight intrusion detection. ACM SIGOPS **45**, 38–53 (2011)
38. Garfinkel, T., Rosenblum, M., et al.: A virtual machine introspection based architecture for intrusion detection. In: NDSS (2003)
39. Kholidy, H.A., Baiardi, F.: CIDS: a framework for intrusion detection in cloud systems. In: ITNG (2012)
40. Modi, C., Patel, D., Borisaniya, B., Patel, H., Patel, A., Rajarajan, M.: A survey of intrusion detection techniques in cloud. JNCA **36**, 42–57 (2013)
41. Patel, A., Taghavi, M., Bakhtiyari, K., Celestino Jr., J.: Review: an intrusion detection and prevention system in cloud computing: a systematic review. JNCA **36**, 25–41 (2013)
42. Gember, A., Krishnamurthy, A., John, S.S., Grandl, R., Gao, X., Anand, A.: Stratos: a network-aware orchestration layer for virtual middleboxes in clouds. arXiv (2013)
43. Chari, S.N., Cheng, P.-C.: Bluebox: A policy-driven, host-based intrusion detection system. ACM TISSEC **6**, 173–200 (2003)
44. Smalley, S., Vance, C., Salamon, W.: Implementing selinux as a linux security module. NAI Labs Rep. **1**, 43 (2001)
45. SUSE AppArmor. https://www.suse.com/support/security/apparmor/
46. Harada, T., Horie, T., Tanaka, K.: Task oriented management obviates your onus on linux. In: Linux Conference (2004)
47. Forrest, S., Hofmeyr, S., Somayaji, A.: The evolution of system-call monitoring. In: ACSAC (2008)