2011 30th IEEE Symposium on Reliable Distributed Systems Workshops

NetODESSA: Dynamic Policy Enforcement in Cloud Networks

John Bellessa[1], Evan Kroske[2], Reza Farivar[1], Mirko Montanari[1], Kevin Larson[1] and Roy H. Campbell[1]
University of Illinois at Urbana-Champaign[1] and University of South Florida[2]
{belless1,farivar2,mmontan2,klarson5,rhc}@illinois.edu and evankroske@mail.usf.edu

***Abstract*—The networking environments found in cloud computing systems are highly complex and dynamic. Consequently, they have strained current policy management and enforcement systems that are based on writing explicit rules about individual hosts. In response, we propose NetODESSA, an inference-based system for network configuration and dynamic policy enforcement. NetODESSA permits the construction of flexible and resilient dynamic networks by allowing network administrators to write general policies about classes of hosts that are combined with runtime information to form network-level actions. Moreover, NetODESSA will infer refinements to the policy from network and host-level data, ensuring that the network remains secure. We have created an initial design for the system and implemented a basic prototype, demonstrating the practicality of this scheme.**

***Index Terms*—Network management, Network monitoring, Resilience, Network security, Dynamic networks**

I. Introduction

Network configuration and policy administration is a manual, time-intensive, and error-prone process. Network administrators are expected to know everything about the hardware and software that define their networks, from the low-level configuration files to high-level network topology and policies involving users and the network at the same time. The continued operation of the network depends on a diverse set of configuration information spread across hosts, switches, and middleboxes throughout the network. Today, centralized network policy management systems such as FSL/FML [6], [7] and Resonance [12] are becoming viable in production networks, simplifying secure network configuration.

Although these new systems address the primary problem of today's network configuration and policy administration methods they fail to address the problems facing tomorrow's networks. They do not consider the problems associated with dynamic networks within clouds of virtual machines. NIST defines cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [8]." Clearly, given the growing need for resiliency in clouds and other complex networks, today's restrictive, static network configuration systems cannot keep up.

We present NetODESSA, a network configuration system for dynamic policy enforcement. NetODESSA extends ODESSA [11], a distributed host-level dynamic policy monitoring system, into the network layer. Network administrators will be able to write dynamic policies that are enforced at the network-level and monitored at the host-level. NetODESSA permits the construction of flexible and resilient dynamic networks by allowing network administrators to write general policies about classes of hosts that are combined with runtime information to form network-level actions rather than explicit rules about individual hosts. Dynamic policies can enable the network to autonomically reconfigure itself, making it more resilient to attacks and system failures while minimizing the need for human intervention.

Following the introduction, we explain technologies related to the implementation of our system in section II. Section III talks about the concepts behind our system, the planned system architecture, and our current prototype. We describe experiments we used to test the system and finally conclude the paper with a brief summary.

II. Background

Our design for NetODESSA makes use of several existing technologies. The first of these is the *Resource Description Framework* with which we make a logical data model to represent the state of the network. Next is *OpenFlow*, an open-source standard for deploying programmable switches that are controlled by commodity PCs. We also use *NOX*, an OpenFlow controller that provides a programmatic interface to the switch. Together, OpenFlow and NOX enable NetODESSA to perform network-level monitoring and flow-level policy enforcement. We describe each of these in more detail below.

A. Resource Description Framework

The Resource Description Framework [3] or RDF is a group of standards for describing resources. It was conceived as a method for embedding meta-information into the web, but it has since been adopted as a universal description language. Objects are described with RDF statements, each containing a *subject*, *property*, and *object*. Several ontology languages

Evan Kroske participated as an intern of the Information Trust Institute at the University of Illinois.

This material is based on research sponsored by the Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

This work was partially supported by the Boeing Company.

978-0-7695-4451-9/11 $26.00 © 2011 IEEE
DOI 10.1109/SRDSW.2011.24

have been built on top of RDF to organize information into hierarchies, include Resource Description Framework Schema (RDFS) [4] and Web Ontology Language (OWL) [2]. When we use a property defined by RDF or RDFS, we prepend it with `rdf:` or `rdfs:`, respectively.

For example, suppose that an ontology contained the RDF statement `(#Alice, rdf:type, #SuperUser)`. This statement says that the resource `#Alice` has a property `rdf:type` with the value `#SuperUser`. If the ontology included another statement `(#SuperUser, rdfs:subClassOf, #User)`, then one could use an inference engine to infer that the statement `(#Alice, rdf:type, #User)` is included implicitly in the ontology, because `#SuperUser` is a subclass of `#User`.

Restrictions are a special type of RDF classes which can be applied to RDF resources based on the properties of those resources. Suppose that we define a restriction `#Staff` which restricts the `#group` property to "staff". If our data model contained the statement `(#Alice, #group, "staff")`, an OWL-compatible reasoner could infer that `(#Alice, rdf:type, #Staff)` belongs to our model.

B. OpenFlow

OpenFlow [9] is a protocol which allows a controller computer to control a compatible switch's behavior by assigning actions to specific types of connections, called "flows." OpenFlow allows switches to defer control decisions to a controller. Flows are defined by a set of fields in packet headers (e.g. source and destination addresses, packet type, physical switch port, etc). Also the flow table associates flows with actions that the switch should take when a matching packet arrives.

When a packet arrives at an OpenFlow switch, the switch checks the packet's header against the flow table. If the packet's header matches a flow in the table, the switch takes the corresponding action. Otherwise, it forwards the packet to the controller, awaiting a decision. Because OpenFlow is an open standard, commodity OpenFlow switches can work with any OpenFlow controller for a wide variety possible uses [9, Section 2].

C. NOX

NOX [5] acts as an OpenFlow controller and provides an API for OpenFlow. It's goal is to present a "network operating system" for building network-level applications. It can manage multiple OpenFlow switches at once, presenting NOX applications with a centralized view of the entire network. Thus, applications are able to effectively control an entire network.

In general, NOX applications are event-driven. During their execution, typically as part of their initialization, applications will register callback methods to be invoked whenever certain events occur. A common event for NOX applications to handle is an incoming packet event which is raised whenever an OpenFlow switch forwards a packet to NOX. We will give a more detailed explanation of how we intend to use events below.

III. NetODESSA

A. Dynamic Policy

To address the shortcomings of existing centralized network management systems, we propose NetODESSA, a system for managing networks with dynamic policy. The primary disadvantage shared by current systems is that they are far too rigid, requiring network administrators to explicitly define rules about specific hosts and static groups or requiring them to define a finite set of states the hosts can be in. To overcome these problems, we present the concept of dynamic policy, a high-level approach to network policy definition.

A dynamic policy is one in which a general high-level policy is defined. Rather than explicitly stating what rules apply to what hosts, network administrators describe what rules apply to hosts displaying certain characteristics. This description, or *base policy*, is fed into an inference engine. Then, using data observed from the network, the inference engine determines how hosts should be classified. Based on these observations, the inference engine refines the policy. The inference engine then uses this *refined policy* to determine if violations are occurring. If a violation does occur, a solution is determined and the proper action is taken. (See Figure 2.)

For instance, consider a secured cloud that houses a repository of sensitive information that is accessible to authorized personnel via SSH. The cloud is hosted internally and can only be accessed from within the local network, including a VPN. Because the repository is not computationally expensive, the cloud's resources are routinely used to preform time-sensitive but non-confidential MapReduce operations.

As a precaution, if the internal network is thought to have been compromised, the cloud will be shutdown. This cannot be done immediately, however, due to the possibility that there are important computations being performed. To ensure that the repository is secured immediately, the network administrator has implemented a dynamic policy which prohibits SSH traffic if the "network health," is not considered secure.

Suppose a laptop that is known to have been stolen establishes a VPN connection to the network. This is caught and the network health is immediately downgraded from "secure," to "compromised." Thus, any SSH traffic is immediately disallowed as it violates the security policy.

B. Terminology

In order to explain our system, we must clearly define our terminology.

- A *policy* is a set of rules that govern network's behavior.
- A *rule* is a $(condition, action)$ pair, such that if the $condition$ is satisfied then the $action$ is taken.
- A *condition* is a set of statements, all of which must be in present in the data model for the condition to be satisfied.
- A *statement* is an RDF statement (See Section II-A). Statement that are present in the data model represent facts about the state of the system.
- An *action* is a change in the network triggered by the inference engine. For example, rerouting traffic through

a middlebox for deep packet analysis or blocking a flow by dropping packets.

C. ODESSA

ODESSA is a distributed host-level policy-compliance monitoring system which forms the inspiration for NetODESSA. The system decomposes rules into pieces that can be checked by individual agents and pieces that must be checked by groups of agents. It then distributes those pieces to the hosts which can check them. An agent reports its host's compliance information to a verifier, which reports policy violations to the network administrator.

ODESSA policies are decomposed into rules that can be monitored at host level, and each rule is distributed only to the ODESSA agents which can detect its violation. Each agent reports the status of its rules to one or more verifiers, which determine whether policy violations are occurring. For example, two agents on the same network may report a pair of events that together cause a violation but, individually, would not. The system resists attacks and data tampering by distributing monitoring responsibilities among the hosts and checking critical rules with multiple verifiers.

A recent variant of ODESSA, called Dora [10], is also worth mentioning. Both ODESSA and Dora have the same goal: distributed policy compliance monitoring. Both accomplish this by decomposing rules into their atomic parts and distributing the work of evaluating compliance to all the members of the system. The primary difference is how this is accomplished. Whereas ODESSA relies on several dedicated verifiers to aggregate results, Dora has no dedicated verifiers; every node can serve as a verifier. Thus, the Dora system is more robust against arbitrary failures and attacks. For our purposes, however, ODESSA provides a more than adequate model to build off of.

Our goal for NetODESSA is to provide ODESSA functionality at the network-level. Using a modified version of ODESSA's inference engine, we will be able to logically infer network policy violations without specifying what they look like. We also use ODESSA's existing host-level monitoring capabilities to complement NetODESSA's network-level monitoring for a more robust set of enforceable policies.

D. Jena

Jena [1] is a Java library for reasoning about RDF. What makes Jena useful is its ability to logically infer new RDF statements from existing statements. Jena allows developers to reason about ontologies of classes, properties, and objects. Like ODESSA, NetODESSA uses Jena to infer whether or not a policy violation exists.

E. Design

Our system design includes four components: the flow sentinel, OpenFlow switches, NetODESSA inference engine, and ODESSA agents. We plan have implemented the two new components, the flow sentinel and NetODESSA inference engine.

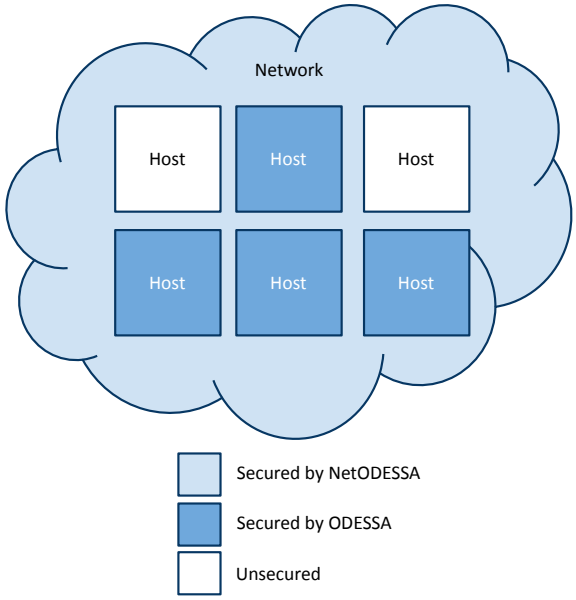

Fig. 1. Network security is ensured by both ODESSA and NetODESSA

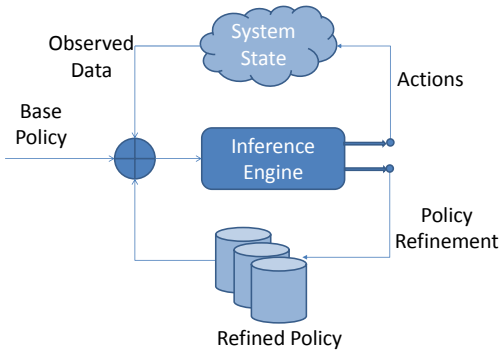

Fig. 2. Information flow in a dynamic policy system

The flow sentinel is responsible for monitoring and controlling network flows. It notifies the NetODESSA inference engine when flows are established for IP traffic and enforce the inference engine's instructions by registering flows with the OpenFlow switches. The flow sentinel is implemented as a NOX application to simplify the management of the switches.

The NetODESSA inference engine is responsible for determining which flows should be allowed and which should be denied, based on the network policy. It uses Jena to infer policy decisions based on the base policy and input from both the flow sentinel and the ODESSA agents. When the flow sentinel sends a new flow to the inference engine, it will check it against its existing policy to determine if it is a violation or if any other special conditions apply to it. Once it has made its decision, it will respond to the flow sentinel with instructions for how to deal with the flow.

```
model = new OntologyModel()
model.read("flow-schema.rdf")
while flowEvent = socket.readFlowEvent():
    if flowEvent.type == FLOW_ADDED:
        model.add(flowEvent.flow)
    else if flowEvent.type == FLOW_REMOVED:
        model.remove(flowEvent.flow)

    violations = model.
        listIndividualsOfClass(Violation)
    for violation in violations:
        log(violation)
```

Fig. 3. The core of the NetODESSA inference engine

F. Flow Sentinel

The flow sentinel is written in Python using the NOX API. This gives us control over the OpenFlow switches on a network. The flow sentinel registers callback methods to be invoked when certain events occur in the network. It listens for flow creation and flow removal events. Flow creation events are raised when a NOX application installs a new flow in a switch's flow table and flow removal events are raised when a flow times out or is otherwise removed.

Depending on the type of event, it will provide various data about the state of the network. Whenever a relevant event occurs, we will parse the data and send it along to the inference engine. To accomplish this, as part of its initialization phase, the flow sentinel will establish a TCP connection to the inference engine, which may or may not be running on the same machine. This also gives the inference engine the ability to invoke changes in the network via the flow sentinel.

G. Prototype

As a proof of concept, we have built a prototype which can detect and log simple violations. The flow sentinel relays flow events to the NetODESSA inference engine which then checks for flows on specific "blocked" ports and logs violating flows to a file. Psuedocode for the inference engine is shown in figure 3.

All of the logic for enforcing policies is contained in the flow ontology and the policy itself. The flow ontology defines two classes: `#Flow`, with properties `#srcIP`, `#srcPort`, `#dstIP`, and `#dstPort`, and `#Violation`, with no properties. The policy can define subclasses of `#Violation` to represent types of violating flows. For example, our first policy defined `#SSHViolation` as a subclass of `#Violation` and a restriction of the property `#srcPort` to the value "22". This made all flows with a `#srcPort` of "22" subclasses of `#Violation`.

IV. Experimental Results

To test the system, we constructed a network with two physical hosts connected by an OpenFlow switch. Each physical host contains 16 virtual hosts. Each virtual host runs a daemon which scans the network for live hosts, picks one at random, and establishes a new flow with that host once per second for 100 seconds.

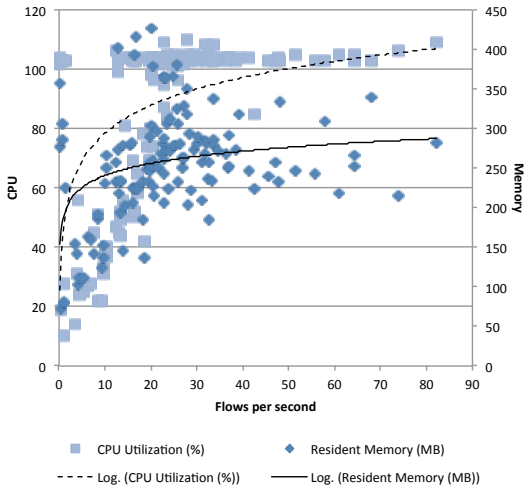

Fig. 4. First experiment results

A. Resource Utilization by Flows per Second

The first test is designed to measure how the memory and CPU usage of our NetODESSA prototype scales with the number of flows per second. The NetODESSA inference engine only checks for traffic on port 22. We measure the resource utilization and number of flows created over 10 second intervals.

As shown in Figure 4, the system's resource utilization scales linearly with the number of flow events per second, saturating CPU capacity at 40 flow events per second. Because we are testing an unoptimized prototype and our system only needs to consider the first packet in each flow, we believe this result indicates that our final NetODESSA implementation will be able to function in production network environments.

B. Resource Utilization by Number of Rules

Our second test measures the resource utilization of the system based on the number of rules enforced. Each rule blocks the use of a single port. For each run, we measures the resource utilization for one minute given a fixed number of rules with the system under load.

Unfortunately, the test results indicate (shown in Figure 5) that our current implementation is far too slow to be useful in real-world network. The CPU utilization is saturated when the inference engine is enforcing only 6. Each query of the data model for violations takes more than a second to run.

We plan to address the performance issues by replacing our prototype's inference engine. We chose to an OWL reasoning engine for the ease with which we could express policies, but such choice led to a high computational cost during the inference process. By replacing our current engine with ODESSA's efficient Datalog inference engine, we expect to solve our prototype's performance issues.

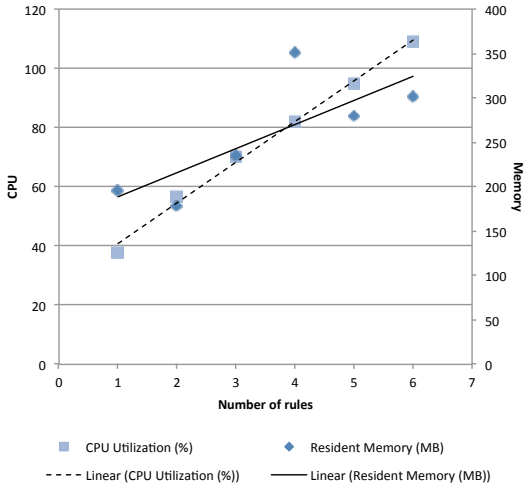

Fig. 5. Second experiment results

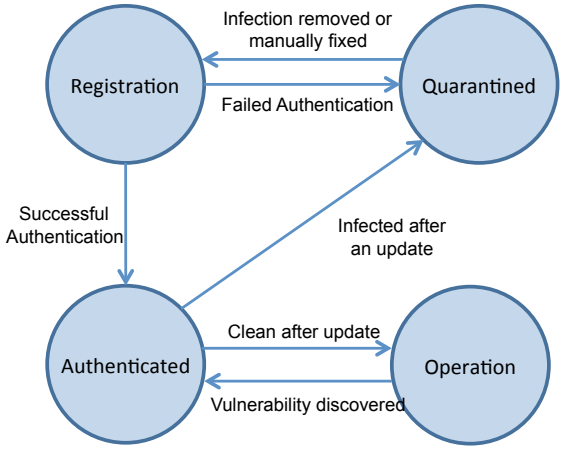

Fig. 6. Host states and transitions in Resonance

V. Related Work

A. FSL/FML

FSL/FML is a declarative network management language designed to simplify and centralize network management for network administrators. It allows administrators to write policies that control the flow of traffic through the network, allowing and denying flows and rerouting packets. High-level policies are compiled into a decision tree which instructs the OpenFlow controller how to handle specific flows. This approach has many advantages over manual network configuration including ease-of-use and security, but it remains focused on defining policies which control the traffic to individual users and hosts. Additionally, the network policy must be recompiled after every change. FSL/FML's static policies cannot describe the resilient network configurations necessary for dynamic cloud networks.

B. Resonance

Resonance is a dynamic network management system for access control. It defines a graph of host states and transitions between those states, ensuring security using a less-general variant of the Bell-LaPadula model. (See Figure 6.) A host has privileges based on its state, and its privileges can change when it transitions to another state. However, Resonance policies must define a finite number of discrete host states. It is impossible for a Resonance policy to dynamically assign privileges to a host. Its restrictive policy requirements limit its utility in dynamic networks.

VI. Conclusion

We are building NetODESSA as a network configuration and dynamic policy enforcement system for the future of cloud-based networks. By accepting high-level, class-based policies and refining them through inference, our system will be more resilient to network changes than existing manual, static, and state-based network configuration methods. By integrating ODESSA's efficient policy monitoring system, we will ensure the performance of the system.

References

[1] "Jena." [Online]. Available: http://jena.sourceforge.net
[2] "Owl." [Online]. Available: http://www.w3.org/TR/owl-features/
[3] "Rdf." [Online]. Available: http://www.w3.org/RDF/
[4] "Rdfs." [Online]. Available: http://www.w3.org/TR/rdf-schema/
[5] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 105–110, July 2008. [Online]. Available: http://doi.acm.org/10.1145/1384609.1384625
[6] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Expressing and enforcing flow-based network security policies," University of Chicago, Tech. Rep., 2008. [Online]. Available: http://people.cs.uchicago.edu/ thinrich/papers/hinrichs2008design.pdf
[7] ——, "Practical declarative network management," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, ser. WREN '09. New York, NY, USA: ACM, 2009, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1592681.1592683
[8] M. Hogan, F. Liu, A. Sokol, and J. Tong, "Nist cloud computing standards roadmap – version 1.0," National Institute of Standards and Technology, Tech. Rep., 2011. [Online]. Available: http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing/StandardsRoadmap/NIST_CCSRWG_092_NIST_SP_500-291_Jul5.pdf
[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746
[10] M. Montanari and R. H. Campbell, "Attack-resilient compliance monitoring for large distributed infrastructure systems," in *NSS*, Milan, Italy, 09/2011 2011. [Online]. Available: http://srg.cs.illinois.edu/srg/sites/default/files/montanari_sec2011.pdf
[11] M. Montanari, E. Chan, K. Larson, W. Yoo, and R. H. Campbell, "Distributed security policy conformance," in *IFIP SEC*, IFIP. Lucerne, Switzerland: IFIP, 06/2011 2011. [Online]. Available: http://srg.cs.illinois.edu/srg/sites/default/files/montanari_sec2011.pdf
[12] A. K. Nayak, "Flexible access control for campus and enterprise networks," Master's thesis, Georgia Tech, 2010. [Online]. Available: http://hdl.handle.net/1853/33813