# XNet: A CAPABILITY ENABLED CLOUD ACCESS CONTROL SYSTEM

by

Joshua Kunz

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

August 2017

**The University of Utah Graduate School**

**STATEMENT OF DISSERTATION APPROVAL**

The dissertation of **Joshua Kunz**

has been approved by the following supervisory committee members:

| | | |
|---|---|---|
| **Jacobus van der Merwe** , | Chair(s) | **June 16th, 2017** Date Approved |
| **Anton Burtsev** , | Member | **June 16th, 2017** Date Approved |
| **John Regehr** , | Member | Date Approved |

by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by **David B. Kieda** , Dean of The Graduate School.

# ABSTRACT

Cloud infrastructures have massively increased access to latent compute resources allowing for computations that were previously out of reach to be performed efficiently and cheaply. Due to the multi-user nature of clouds, this wealth of resources has been ”siloed” into discrete isolated segments to ensure privacy and control over the resources by their current owner. Modern clouds have evolved beyond basic resource sharing, and have become platforms of modern development. Clouds are now home to rich ecosystems of services provided by third parties, or the cloud itself. However, clouds employ a rigid access control model that limits how cloud users can access these third-party services. With XNet, we aim to make cloud access control systems more flexible and dynamic by modeling cloud access control as an object-based capability system. In this model, cloud users create and exchange ”capabilities” to resources that permit them to use those resources as long as they continue to possess a capability to them. This model has collaborative policy definition at its core, allowing cloud users to more safely provide services to other users, and use services provided to them. We have implemented our model, and have integrated it into the popular OpenStack cloud system. Further, we have modified the existing Galaxy scientific workflow system to support our model, greatly enhancing the security guaranteed to users of the Galaxy system.

To my family, friends, and fellow students.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

While cloud infrastructure has massively increased access to large-scale compute resources, by allowing dynamic provisioning by multiple users, it has introduced new security challenges not faced in traditional "enterprise-like" deployments. In a standard enterprise datacenter environment, it can usually be assumed that all resources contained within have some level of mutual trust, or at least have common incentives, specifically, to achieve the goals of the datacenter owner. If an enterprise datacenter is secured from external threats, it can be assumed that there will be no purposeful internal threats. As such, enterprise networks make little effort to isolate their resources internally. Public-facing servers with large attack surfaces may be isolated from, say, all internal devices on the enterprise's network, but the internal devices may not be isolated from *each other*. This is due to the fact that, by default, networks implement an *Ambient Authority* security policy. Connectivity is "on" by default, and must be restricted using Internet protocol (IP) routes, firewalls, or other means. Due to this, administrators typically don't take the extra effort to isolate internal devices from each other, instead opting to isolate internal resources from external ones. This creates a catastrophic failure mode for the compromise of a single internal device. Once an attacker is able to gain leverage in the internal domain, they have free reign to attack all other subsystems.

Unfortunately, the transition to cloud infrastructure has not changed this aspect of enterprise access control. Clouds provide their tenants with "user" and "project" abstractions that model, in the cloud, a typical enterprise domain: a set of resources connected by an Ambient Authority network with strong isolation from the "outside world" through firewalls. We argue that replicating the enterprise access control model in a cloud is wrong on two fronts. First, it retains the property of ambient network authority that is present in enterprise networks. Instead, we advocate the Principle of Least Authority (PoLA),

where resources begin with no network connectivity and must be *granted* the right to communicate from an administrator. This helps mitigate the catastrophic failure mode described above, by requiring the administrator to explicitly model the authority of each resource over another. Second, it limits the ability of cloud tenants to interact with each other, by placing them in walled gardens. By collocating many different workloads in a single datacenter, it becomes feasible for appliance providers to directly provide their services to other cloud tenants. This is common on clouds today [14, 24, 29], with some clouds even providing explicit support for it. However, the types of services are limited by the abstractions cloud providers supply for inter-tenant interactions. Most such services are supplied as either Virtual Appliances as a Service (VAaaS) where an appliance provider gives a customer access to a prepackaged Virtual Machine (VM) that the customer can then deploy in their cloud, or what we call "remote services" where the customer is expected to use an Application Programming Interface (API) provided by the service provider. In the first (VAaaS) case, the service provider must be able to package their services as a VM (which cannot easily be done), and trust the user with the software contained within, which may be proprietary. In the second case, the customer must trust the remote provider not to steal the data supplied to their API. We argue that by carefully relaxing the isolation between tenants, many new forms of tenant-tenant interaction, such as mutually-distrusting services, can be realized.

To this end, we propose XNet. XNet is a cloud extension that adds an object capability system to an existing cloud. Capabilities are an access control mechanism that allow for flexible control over the access control policy. Resources in the system are modeled as "objects" and the right to manipulate the resource associated with an object is represented as a pointer or capability to one of these resource objects. The access control policy is described by the current distribution of capabilities to users in the system. The capabilities can be exchanged between users, allowing for flexible control over not only what the access control policy is, but who controls the policy. In XNet, we use capabilities to construct an additive security policy framework. There is, by default, no connectivity between cloud resources; all connectivity must be constructed by manipulating the policy graph, thereby enforcing the Principal of Least Authority. Since these policy control capabilities can be exchanged, we can leverage them to accomplish the type of cross-tennant collaboration

described earlier.

## 1.1 Thesis Statement

Augmenting current cloud access control systems with capabilities enhances functionality and security of real-world applications, with low run-time costs.

We have developed an implementation of XNet based on the OpenStack cloud system. We have constructed a Software Defined Networking (SDN) controller that integrates with the OpenStack networking driver interface, allowing us to create "capability enabled" networks in OpenStack. We have also created some basic example applications that illustrate the types of collaboration possible with the XNet framework. For example, we have a working example of a secure collaborative hadoop instance. Finally, to show that XNet can be used to add security to existing real-world systems, we have extended the Galaxy [2] scientific workflow system to use XNet. This adds significant security over the existing system, while maintaining most of the remaining functionality.

Specifically, our contributions are as follows:

- The application of an object capability model to cloud access control, where cloud resources (such as VMs, and network connectivity) are represented by objects, and capabilities are used to control them.

- The construction of a "mutual isolation" protocol based on our cloud capability model, as well as a "Membrane" object inspired by existing research on capability systems. This protocol enables novel tenant-to-tenant services on XNet-enabled clouds.

- A prototype implementation of our cloud-specific object-capability that is capable of enforcing policies defined in the model on SDN-enabled networks.

- A XNet-enabled cloud that integrates our prototype controller with the OpenStack cloud orchestration system.

- A set of XNet-aware extensions to the Galaxy scientific workflow system that meaningfully extend the security of the Galaxy system, specifically, allowing scientists to keep their data private while using the system.

- Several microbenchmarks of the XNet controller's overhead, showing it adds sub-millisecond overhead to most operations.

# CHAPTER 2

# BACKGROUND

XNet's core security model is based around an object capability system. Such systems have existed for many years, and have been used extensively in the Operating Systems [10, 12], and programming languages [13, 15] literature to implement access control policies. The XNet system builds on these previous systems, using concepts they describe to model a cloud-specific access control policy. Since concepts from this previous research are used extensively by XNet, we'll begin by reviewing the basic model they propose.

An object capability system is composed of an object-capability graph. *0bjects* are pieces of code and data that have methods that can be executed by other objects who possess *capabilities*, which act as references to other objects. These capabilities model the **privileges of the owner** (or *principal*, in the capability literature) over the object they point to. Capabilities are unforgeable. To obtain a capability for an object, a principal must have it granted to them by the system itself (for example when new objects are created) or by another principal in the system. By modeling cloud resources as objects in the capability system, we can model concepts like VM ownership and network connectivity as capabilities. Owning a capability to a special "Flow" object, for example, entitles the owner to send network traffic to another, specific, VM. The capabilities a principal owns can therefore describe the flexible access "policy" of that principal.

In XNet, capabilities are exchanged between principals using channels called "rendezvous points". By representing the mechanism of capability exchange itself as an object, XNet also describes a "policy policy" or "meta-policy" that details how principals are able to manipulate the policy directly. Further, since these capabilities convey extremely fine-grained privileges over the access policy and meta-policy, XNet can allow users to construct novel, highly dynamic, policy management systems.

# CHAPTER 3

# ARCHITECTURE

XNet can primarily be understood as a network control architecture. XNet integrates with the cloud system, but it is largely independent from it.

The architecture of the XNet system is depicted in Figure 3.1. The figure shows two "views" of the XNet system, an abstract-network representation, and a high-level architectural diagram. The arhitectural diagram on the right-hand side of the image, represents the architecture of the XNet system itself, while the left-hand side of the figure shows the types of network policies that can be implemented using the XNet system.

The XNet architecture consists of three major components: the cloud controller, the XNet capability-enabled network controller, and the set of capability-aware nodes that make up the network controlled by the XNet controller. The cloud controller is responsible for most cloud-related tasks: managing cloud users, receiving requests to create virtual resources, communicating with the physical nodes to spin-up new VMs and tear down VMs that are no longer desired or needed. However, instead of managing the cloud infrastructure network directly, the cloud controller communicates with the XNet network controller. As new virtual compute resources (nodes) are created, the cloud controller informs the network controller of their network location (switch,port) and pertinent metadata, like IP and MAC address. The network controller uses this information to manage the capability system. As new nodes are added, the network controller adds them as objects to its internal representation of the system. These nodes can perform operations "on behalf" of their equivalent virtual objects by using a capability protocol that is exposed by the network controller to the nodes. As the state of the internal objects changes (due to operations executed by the actual nodes) the controller maps those changes into physical network changes that are propagated onto the managed networks. For example, if the capability system's state changes to allow one of the internal nodes to receive traffic on

TCP port 80, then the controller would configure the actual cloud network to allow TCP port 80 traffic to the actual node associated with the internal node object.

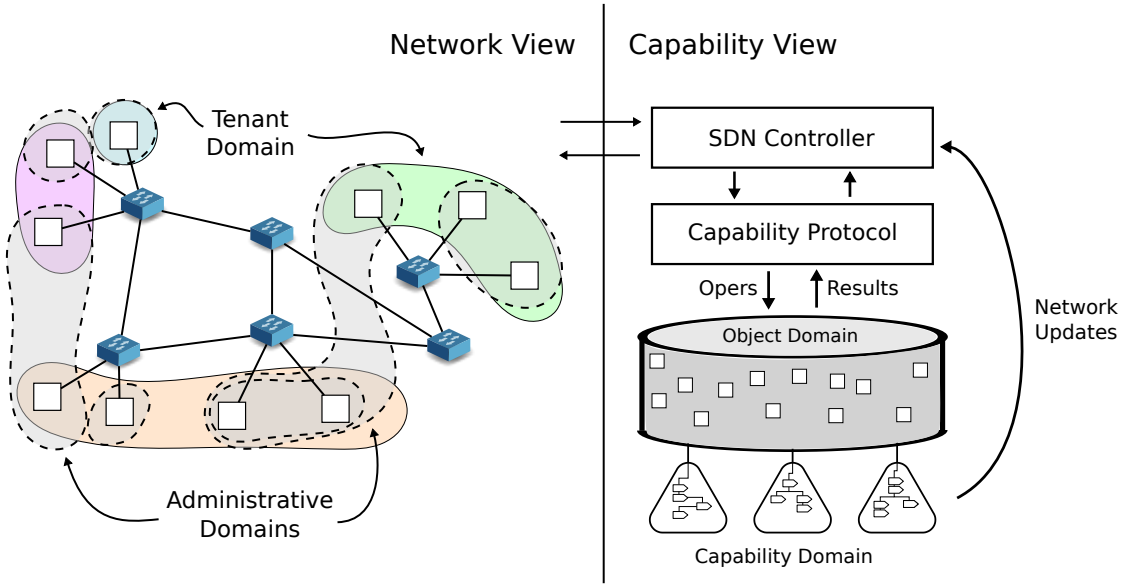


**Figure 3.1**. High-level XNet architecture showing the network vs. capability system-level views.

# CHAPTER 4

# THREAT MODEL

XNet is primarily an access-control framework. XNet aims to preserve the isolation capabilities of current clouds, while allowing tenants to weaken that isolation if they choose, to allow collaboration between cloud tenants. The current configuration of the XNet system is a definition of the current access-control policy for the network. This policy begins in a maximally isolated state, where are nodes are biconnected, and cloud tenants only have the rights to define the policy for their own resources. XNet is primarily concerned about three potential types of attacks:

1. Attacks that allow communication that violates the policy to occur.

2. Attacks that *prevent* communication authorized by the policy to occur.

3. Attacks that allow the policy to be changed in a way that violates the XNet model.

We assume that the XNet-enabled cloud provider, and the cloud provider's infrastructure (switches, physical machines, etc.) are trustworthy. We also assume that the cloud infrastructure is not vulnerable to attacks based on the physical constraints of the system. For example, XNet does not mitigate Denial of Service or side-channel attacks. We additionally assume that the cloud provider's implementation of XNet is correct. That is to say it correctly implements the model as described in Chapter 5.

Therefore, we assume our attacker is attached to the network infrastructure controlled by the XNet-enabled cloud. For example, an attacker may be in control of a virtual machine attached to the XNet-managed infrastructure. The attacker may send and receive traffic, including XNet-protocol messages to the XNet controller; it may even try and forge XNet-protocol messages.

The model described in Chapter 5 describes the set of valid policy change operations. Given the assumptions in this threat model, we can trust that these changes will operate

as described. Any policy-change-protocols built on the operations described in the model will therefore be correct, as long as they correctly utilize the operations described in the model. Therefore, the threat model does not address attacks against the protocols, other than by assuming that the model, as described, is implemented correctly.

# CHAPTER 5

# MODEL

The XNet capability model consists of a set of objects (Table 5.1) that are used to model the cloud and the distribution of capabilities, as well as a set of operations (Table 5.2) that enable basic interaction with capabilities and the object system. The operations are executed by principals; any capability operations are performed "in their context" so to speak. A principal cannot execute a capability operation using a capability they do not own.

## 5.1 Basic Capability Distribution

A principal in XNet begins its life with no capabilities, and therefore, cannot access any objects. Instead, it only has access to a set of basic "operations" (detailed in Table 5.2). Two of these operations, `create` and `rp0`, connect the new principal with the rest of the capability system. The first of these operations, the `create` interface, is used to create new objects. When a principal creates an object, they receive a capability to the newly created object as a result of the operation. The caller selects the type of the created object, by passing it as an argument to the `create` operation. The second of these operations, `rp0`, is invoked with no arguments, and returns a capability to a special rendezvous point object. Rendezvous point objects are the main mechanism of capability exchange. They act as channels through which capabilities can be "passed". They implement a `send` and `recv` interface. The former, when given a capability, will add it to the rendezvous point's internal queue; the latter will then pop the least recently added capability off of this queue and return it to the caller. Together they implement a first-in-first-out capability channel. Principals invoke these methods, like all methods, by using a third operation: `invoke`. Invoke takes a capability to an object and the method on that object to invoke, as well as any arguments (such as the capability to send for the `send` method of rendezvous points).

In XNet, all capabilities are only owned by a single principal. As part of this exchange,

a new capability is created that references the same object as the original capability. We say that this capability is "derived from" the original capability. If the derived capability is then sent and received itself, the resulting capability will be derived from the derived capability itself. Further, the original capability could be sent and received again using a rendezvous point, resulting in two derived capabilities. As such, these capabilities form a tree of derivation. In the literature, these are called Capability Derivation Trees (CDTs). By modeling capability exchange as a series of derived capabilities, XNet is able to implement additional capability management operations.

The most basic of these capability management operations is `delete`. When a capability is deleted, it is removed from the set of capabilities the principal controls, and it is deleted from the CDT. Derived capabilities are not removed. This allows a single principal to de-escalate their privileges without affecting the privileges of others. As an extension of this, XNet also provides the `revoke` operation. This operation executes `delete` on every *decendent* of capability passed to it. This operation allows the caller to "reclaim" their authority over a capability from any other principal to whom they have given it. Since it only applies to the descendants of a given capability, sub-trees of the CDT can be revoked independently.

A final operation is provided, `mint`, which implements a basic capability-duplication operation. `mint` takes a capability as an argument, and returns a capability to the same object that is a child of the originally capability in the CDT. It has the same effect as if the principal was to create a new RP, and then send and recv a capability using it.

## 5.2 Modeling the Cloud

In order to allow users to manipulate cloud resources in the capability domain, XNet models cloud resources as capability objects: `NodeOwner`, `NodeLease`, and `Flow`. The interfaces for these objects, along with all other objects, are listed in Section 5.3. To achieve strong isolation and collaboration between mistrusting parties, XNet implements a two-tiered model of node ownership and control (see Figure 5.1). For every node, XNet models a *single* `NodeLease` object. A capability to this object gives the owner effective control over the node. This object implements methods for every operation that can be performed in XNet including the ability to invoke methods on object. These methods are executed "in

the context" of the modeled node. That is to say, they are executed as if they were executed by the modeled node. When a `NodeLease` is destroyed, the associated node is wiped clean. All capabilities owned by the node are `delete`'d and all flows that allow sending packets *to* the node are destroyed, re-isolating the network access policy of the node. Finally, the node itself is re-booted, so any state stored in the memory of the node is destroyed as well.

Despite the fact that a node is only ever controlled by a single party at a time, the node may still have multiple *owners*. We say a principal has ownership if they possess a capability to the `NodeOwner` object associated with the node. In the XNet model, ownership gives the owner the privilege to, at any time, acquire a new `NodeLease` to the associated node. They do this using the `reset` method of the `NodeOwner` object. This method destroys the old `NodeLease`, creates a new one, and returns a capability to the newly created lease to the caller. This reset mechanism is very useful for protocols between mutually distrusting parties. One party can grant a capability for a `NodeOwner` to the other untrusted party. This capability can only be used to reset the underlying node, the first party doesn't have to worry about capabilities owned by the node leaking from the `NodeOwner`. Additionally, once the untrusted party has received the `NodeOwner` capability, it can use it to reset the underlying node. This guarantees to the second party that the original owner no longer has access to any capabilities owned by the original node. The node has been completely separated from its original owner.

### 5.2.1 Interfacing with Cloud Authority

XNet is only concerned with the definition and enforcement of access control policy at the level of cloud resources. Importantly, it does not model the cloud interfaces that are used to create and destroy cloud resources. XNet relies on the cloud's standard resource creation system to correctly manage who can create what resources, and who can destroy what resources. To allow a newly created node to enter the capability-enabled world, it must know to whom the node belongs. Due to this, XNet must map the cloud's understanding of a resource's "owner" to a principal in the capability domain. To this end, XNet introduces the concept of a "master workflow agent". The master workflow agent is modeled as a node that is associated with some cloud-level domain, like a project. When a new node is created in a project, the master workflow agent for that project is delivered a

`NodeOwner` capability for the newly created node object. From this capability, all ownership and control over a node is derived. This workflow agent acts as the "root" of authority for all nodes created in that project. It must act to begin *any* capability exchange between nodes in the same project.

## 5.3 Modeling Collaboration

A primary goal of the XNet system was to allow the tenants on the cloud to create inter-tenant policies that facilitated workflows that are insecure using current cloud systems. The objects and operations above hint at some form of inter-tenant collaboration. For example, one tenant could give another a capability, then revoke it after some period of time, or a tenant could reset a node to isolate it from its original owner. However, capabilities can only be exchanged by nodes in the same tenant. There is no capability channel that "crosses" multiple tenants.

### 5.3.1 Broker

To facilitate the communication of capabilities across tenants, XNet gives every master workflow agent a capability to a `Broker` object. This object exposes a simple two-method interface: `register` and `lookup`. Using the register method, a tenant can pass a name and a capability to be associated with that name. By communicating the name with another tenant using an out-of-bound channel, the other tenant can use the `lookup` interface to obtain a capability to the object that was registered using the same name. We envision service-providers using brokers to register "service" rendezvous points that will allow multiple clients to initiate service protocols and exchange capabilities, thereby "breaking" the full isolation between tenants. Therefore, the broker acts as a sort of "super-root" capability, a capability that is shared by every tenant, and can act as a communication channel between them.

### 5.3.2 Annotated Capabilities

In XNet, capabilities can be tagged with *annotations*. These annotations are added to describe the capability's lifetime and privileges. We represent them as a set of triples of the form $\langle L, A, P \rangle$ where $L$ is the object this capability's life is attached to, $A$ is the object that is capable of removing this annotation, and $P$ is itself a set of allowed methods. Typically $A =$

$L$, meaning that the object associated with the lifetime of the capability is often the same capability that is responsible for removing the annotation. Restrictions on the methods that can be invoked are done using $P$. These annotations are associative and commutative; the order they are added and removed does not matter. The full set of enabled operations is defined by the intersection of all $P$s in every annotation. When a capability is returned from a method that was invoked using an annotated capability, the result is the original returned capability with its original annotations unioned with the set of annotations that were on the object used to invoke the operation.

#### 5.3.2.1 Membranes

Membranes are an object that is used to model the "domains of control" of different tenants. Consider the case where two tenants $A$ and $B$ want to cooperate on a data analysis problem. $A$ has some secret data it doesn't want to give to $B$, but $A$ doesn't know how to set up the data-analytics cluster required to efficiently perform the analysis. Using the operations and objects described above, $A$ has a few options. First, $A$ could reset the nodes before giving them to $B$ (preventing $B$ from accessing any data stored on them) and reset them after getting them back, but that would erase any configurations that $B$ did to the nodes . Second, $A$ could grant $B$ access to the nodes, and then use the `revoke` operation later to remove $B$'s access. However, this doesn't work either, as all of the capabilities $B$ creates will "descend" from the original set of capabilities $A$ gave to $B$, again, undoing any configurations $B$ has made to the policy.

Instead, we need a middle ground: a mechanism that will allow $B$ to manipulate the policy, but allow $A$ to verify, at some point, that $B$ no longer has the ability to manipulate the policy, without destroying the parts of $B$'s policy that $A$ wants to keep. Membranes provide this abstraction. Each membrane portions capabilities into two halves: the "inner" half, and the "outer" half. By default, all capabilities are on the "inner" side. Capabilities can be sent to the "outer" side by passing them to `membrane.wrap`. This function returns a new capability with the annotation $\langle M, M, FULL \rangle$, where $M$ is the membrane object we executed the `wrap` method on, and $FULL$ is the full set of capability privileges (no restrictions). Due to the annotation rules, this annotation will be added to any new capabilities that are created from these "outside" capabilities. However, when a capability with such

an annotation is passed back to the `membrane.wrap` method of the same membrane, the annotation is removed.

Membranes can also be used to wrap rendezvous points, which will implicitly invoke "m.wrap" on `send` and `recv`. These wrapped rendezvous points act like "channels" between the inner and outer worlds. Anything that passes from inner to outer gets "wrapped" and anything that passes from outer to inner gets untagged. Then, at any point, the party that possesses a capability to the membrane object can execute `membrane.clear`. This method destroys the membrane, and therefore any capabilities that are annotated with the membrane's annotation (since those capabilities have the same lifetime as the membrane). Anything that has passed back through the membrane before the membrane is cleared will be spared. This operation can be though of as a "selective revoke". In fact, if capabilities never get passed back through the membrane (never call "membrane.wrap" on a capability annotated with the membrane's wrap annotation), `membrane.clear` functions identically to the `revoke` operation.

A critical feature of membranes is that they provide a two-way guarantee. They destroy all capabilities that have only passed one direction, regardless of whether that is from the inside to the outside, or the outside to the inside. This means the membranes can be used to detect whether or not a capability $c$ is a child of some other capability $p$. All we need to do is invoke `membrane.wrap` on $p$, then later when we want to verify that $c$ is a child of $p$, we call `membrane.wrap` on $c$, and then invoke `membrane.clear`. If the capability was a child of $p$, it will still be annotated with the membrane annotation. Invoking `membrane.wrap` will *remove* the annotation, sparing it from deletion when the membrane is cleared. If $c$ is not a child of $p$, a wrap will be added to the resulting capability. Therefore, when the membrane is cleared, the capability will be destroyed. This feature is critical to the construction of mutually-distrusting collaboration protocols, as we cannot trust the other party not to lie about the lineage of a capability.

#### 5.3.2.2 Sealers and Unsealers

Using membranes coupled with reset, we can create nodes that are completely isolated from all other nodes. By introducing sealer/unsealers, we can re-establish connectivity with such isolated clusters through mutual agreement. Sealer/Unsealer objects imple-

ment two methods: `su.seal` and `su.unseal`. When a capability is passed to `su.seal`, a new capability to the same object is returned, with the annotation $\langle S, S, \{\}\rangle$ where $S$ is the Sealer/Unsealer object. This annotation prevents any methods from being executed using this capability. A capability with such an annotation can later be passed to the Sealer/Unsealer's `unseal` method, which will return a new capability without the annotation, effectively "unlocking" the capability, making it usable. By constructing a protocol where sealed capabilities have to pass through multiple principals, each principal can seal the capability with their own sealer/unsealer. Since the privileges only ever restrict, the capability will be unusable until all the seals are removed, and they can be removed in any order.

To enable these protocols, our model explicitly prohibits membrane wraps from being added to capabilities to Sealer/Unsealer objects themselves, since these objects do not themselves allow the passage of capabilities (and sealed capabilities keep all of their additional wraps, so sealing a capability will not prevent it from being deleted after the `membrane.clear` method is invoked on a sealed and wrapped capability).

### 5.3.3 Building Blocks of Collaboration

When used together, the operations and object described above can form powerful isolation primitives or "blocks". These blocks can be composed into even more powerful "protocols" as described in Chapter 6. In this section, we describe these basic building blocks.

#### 5.3.3.1 Permanent Shared Administration

This building block is the simplest. There are two nodes $A$ and $B$ that are owned by different tenants, who want to share administration of a third node $C$ that is owned by $A$. $A$ just grants $B$ a capability to the `NodeLease` object for $C$, and that's it. Both $A$ and $B$ can define the policy, forever, or at least until $A$ revokes $B$'s capability to the `NodeLease`.

#### 5.3.3.2 Flexible Shared Administration

This building block builds on the permanent shared administration block, by adding time-based or "flexible" shared administration. By granting a capability, and then later revoking it, two nodes can share administration for a short-period of time.

#### 5.3.3.3 Single-Way Isolation

Using node capabilities and the `reset` method, a tenant can "lend" a node to another untrusted tenant. Consider two tenants *A* and *B*. If *A* wants *B* to perform some computation on behalf of *A*, *A* can send *B* the node capabilities for some set of nodes that *B* can use by first calling the `reset` method on them, which will ensure that *A* can no longer communicate with them in a networking sense and in a capability sense. *B* can safely give these newly reset nodes capabilities to resources that are essential to the computation, but that are not sharable with *A*. This allows *B* to isolate its data and computation from *A*, but it doesn't allow *A* to do the same to *B*. *A* still must trust *B* with its data, since *B* has an unlimited ability to manipulate the nodes *A* gave it. Further, since this isolation is created through the capability system, it can be "chained" like any other capability. *B* can pass the node capability to another provider *C* who can in-turn reset the node to isolate it from both *B* and *A*.

#### 5.3.3.4 Two-Way Isolation

To achieve two-way isolation, we use the same steps as in the single-way isolation case, with the addition of a `Membrane` object. By sending the node capabilities to *B* through a membrane, *A* can ensure that after the membrane is `cleared` *B* can no longer connect to any of the nodes. Also, since *B* reset the nodes after receiving them from *A*, it can be sure that *A* doesn't have any access to the nodes that *B* has not granted to *A*. Now, *A* still possesses node capabilities to the nodes, so it has the ability to reset them, but this does not grant *A* access to any capabilities or information that *B* has given those nodes. It may break the system that *B* has set up, but won't cause any information or privileges to "leak out". Similarly to the previous case, this can be chained. Inside of an isolated cluster, the nodes can perform two-way isolation again to form even finer granularity isolation bubbles. Using this building block, we can construct any tree-like isolation control policy, but this does not suffice for some workloads.

#### 5.3.3.5 Controlled De-Isolation of Two-Way Isolated Nodes

To enable the construction of full graphs of isolated bubbles, we can use `SealerUnsealer` objects. The rules about membrane annotations do not apply to `SealerUnsealers`, so they can persist after the membrane is cleared. When we're setting up a two-way isolated

cluster, instead of directly giving the other party access at the end of the setup, we can give them *sealed access*, by first applying the `su.seal` method to the capability. By doing this, we ensure that connectivity to the isolated cluster can only be established with the cooperation of the owner of the `SealerUnsealer`. The owner can then create a second two-way isolated cluster, and the ”outer” party of the isolation can pass the sealed capability between the clusters. This can only be accomplished through *mutual agreement* of the parties. Then, using the key that exists in both clusters, the access can be “unlocked” and the sealed clusters can be joined by exchanging capabilities.

**Table 5.1**. A full listing of capability objects and their methods.

| |
|---|
| **NodeOwner** |
| `NodeLease node.reset()` |
| **NodeLease** |
| `node_lease.create(...)` |
| `node_lease.rp0(...)` |
| `node_lease.delete(...)` |
| `node_lease.revoke(...)` |
| `node_lease.mint(...)` |
| `node_lease.invoke(...)` |
| **Flow** |
| **RendezvousPoint** |
| `void rp.send(cap c, string msg)` |
| `(cap, string) rp.recv(int timeout)` |
| **Membrane** |
| `cap m.wrap(cap c)` |
| **SealerUnsealer** |
| `cap su.seal(cap c)` |
| `cap su.unseal(cap c)` |
| **Broker** |
| `void b.register(string name, cap c)` |
| `b.lookup(string name, int timeout)` |

**Table 5.2**. XNet operations.

```
object create(object_type type)
void delete(cap c)
invoke(cap c, method_name m, args...)
cap rp0()
cap mint(cap c, [specification spec])
void revoke(cap c)
```

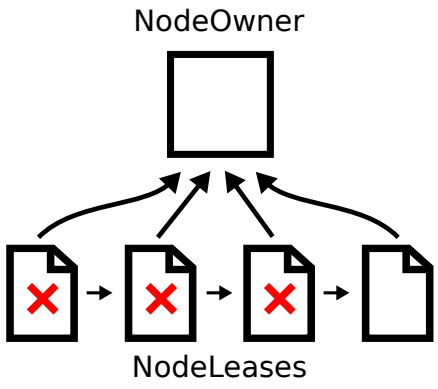


**Figure 5.1**. NodeLease objects modeling the lifetimes of a NodeOwner object

# CHAPTER 6

# EXAMPLE PROTOCOLS

In this chapter, we detail an example isolation protocol that can be constructed using the XNet model: Application as a Service. The Application as a Service protocol is designed to address a common collaboration problem in modern clouds. Today, there is no way for a service provider and customer who are mutually distrusting to cooperate. Either the customer must trust the service provider with its data, or the service provider must trust the customer with its application or data. Sometimes this is not even feasible. XNet addresses this problem by using the Two-Way Isolation building block as described in Section 5.3.3.4.

## 6.1 Application as a Service

Application as a Service is a protocol between two parties, a customer and a provider. We'll denote them as $C$ and $P$. The goal of the protocol is to allow $C$ to use a service provided by $P$ without $C$ and $P$ trusting each other. Specifically, we will create a cluster of nodes $P'$. Intuitively, this will be an isolated cluster of provider ($P$) controlled nodes that can communicate with $C$ but cannot communicate with $P$. Additionally, since we don't want $C$ to be able to connect *arbitraily* to $P'$, the connectivity between $P'$ and $C$ will be created by $P$. So, assuming that each party is rational, self-interested, and distrusting, they will develop the minimal amount of connectivity required to realize the service provider $P$ is trying to provide.

For simplicity of explanation, we will assume that a few capabilities exist. We assume that both $C$ and $P$ own a capability to some rendezvous point. We call this the service rendezvous point. It is the communication channel over which the protocol is performed. Further, we assume that $C$ has capabilities to a set of Node objects. We'll call this set of capabilities $C_{nodes}$.

The pseudocode for this protocol can be found in Figures 6.1 and 6.2. It consists of

two major pieces, one for the customer to execute, and one for the provider to execute. The customer code can be found in Figures 6.1, and the provider code can be found in Figures 6.2. The customer initiates the protocol by invoking request_service with the service rp as an argument, and another rp, the node rp, on which the customer has already sent capabilities for the nodes that will become the $P'$ cluster. Before sending the capabilities to these nodes, $C$ creates a membrane, and wraps them with the membrane. As described in Section 5.3.2.1, this allows $C$ to later remove all capabilities $P$ has to these objects. Since rendezvous points gracefully block until there are capabilities available, the provider has already called receive_service_request earlier with the shared service rp. It additionally takes a rendezvous point on which the nodes for the request will be stored. Once receive_service_request begins receiving node capabilities, it resets them and then sends them on the supplied node rp. The reset is important. It guarantees to $P$ that the nodes also no longer have connectivity to any node in $C$. Therefore, $P$ now knows that it can safely transfer its proprietary service logic and data onto the nodes without fear that $C$ will be able to exfiltrate it. After the nodes have been received and reset, $P$ configures the cluster with the service it will be providing to $C$. As part of this configuration, $P$ creates a rendezvous point on the node that will serve as a frontend for the service. This rendezvous point will be how all connectivity between $C$ and $P'$ is created. To finalize the service setup, the created rp is sent back to $C$ over the service rp.

Finally, the customer receives this resulting rp, and then clears the membrane. At this point, the protocol is finished, and the security guarantees are established. Due to the membrane, no node in $P$ can communicate with $P'$. Only $C$ holds a capability to the rendezvous point in $P'$. However, since the nodes were reset, $C$ holds no capabilities to any objects (for example nodes) in $P'$ except for the rendezvous point it just received. It can only obtain capabilities to the nodes in $P'$ according to the logic in $P'$, which was defined by $P$, who has established a secure frontend for the service it is providing in $P'$. It is also important to note that $P'$ is in fact composed of the nodes that $C$ sent to $P$, as $P$ could potentially try to de-isolate $C$'s data by creating a cluster from nodes that did not pass through the membrane (and therefore are still connected to $P$ after the membrane is cleared). However, this is not possible. Since $P$ sent the result rp back through the membrane, if it was not already wrapped by the membrane, it would have been destroyed

when the membrane was cleared and the protocol would have failed (*C* would not have been able to use the rp). This dual-sided nature of membranes is critical to the operation of this protocol. It guarantees not only that the original nodes are isolated, but that they are in fact the same nodes *C* originally gave to *P*. At this point, *C* can be sure that *P′* is in fact isolated from *P*, and there is no way that *P* can exfiltrate *C*'s data. *C* can proceed using the service as it desires.

```
cap request_service(cap service_rp, cap node_rp):
    membrane = create(Membrane)
    wrapped_rp = membrane.wrap(service_rp)
    while not node_rp.empty():
        wrapped_rp.send(node_rp.recv())
    result_rp = wrapped_rp.recv()
    membrane.clear()
    return result_rp
```

**Figure 6.1**. Application as a Service customer's code

```
cap receive_service_request(cap service_rp, cap node_rp):
    while not service_rp.empty():
        node = service_rp.recv()
        lease = node.reset()
        node_rp.send(node)
        node_rp.send(lease)

void finalize_service(cap service_rp, cap result_rp):
    service_rp.send(result_rp)

void full_service(cap service_rp):
    node_rp = create(RendezvousPoint)
    receive_service_request(service_rp, node_rp)
    # Use the values in the node_rp to set up our service and configure the
    # system...
    result_rp = ...
    finalize_service(service_rp, result_rp)
```

**Figure 6.2**. Application as a Service provider's code

# CHAPTER 7

# IMPLEMENTATION

We have implemented a proof-of-concept of the XNet capability-enabled cloud based on the OpenStack cloud controller. The implementation consists of three parts: A set of OpenStack extensions that allow the XNet network controller to operate as the network controller in OpenStack, a capability-enabled network controller that implements the capability model, and a client library and wire protocol that allow the capability-enabled clients to interact with the XNet network controller.

## 7.1 OpenStack Integration

The OpenStack integration is mainly composed of an XNet OpenStack network driver implemented using the ML2 network interface. It is largely based on the OpenVSwitch driver [3] that is commonly used in OpenStack. This is the XNet controller's primary method of interaction with the OpenStack cloud controller.

As new VMs are created, the OpenStack compute controller informs our XNet driver that a new virtual port has been added, along with the network location of this new virtual port. The XNet driver then plumbs the virtual interface into the local OpenVSwitch instance (that is shared by all virtual machines on a single physical node) and tells the XNet controller about this newly created virtual interface. This not only includes the physical (switch, port) location of the new node, but its identifying network information as assigned by OpenStack (its MAC and IP in our implementation) as mentioned in Chapter 5. The XNet controller then acts as the OpenFlow controller for this local OpenVSwitch, pushing flows onto and popping flows off of the switch as appropriate.

In addition to this network driver, we extend the OpenStack project API to include a flag that toggles whether or not a project will be managed by the XNet controller or the standard OpenStack management scheme. If a project includes the "capability-enabled" flag, it is also expected to pass a python program that will act as the "Master Workflow-

Agent" for that project. The master workflow agent is modeled similarly to a Node. It is run in a network namespace on the OpenStack that has a single interface connected to a local OpenVSwitch that is controlled by the XNet controller. It communicates with the XNet controller using the same API that is used by nodes on the system. When it is created, the XNet OpenStack network driver informs the XNet controller using the same mechanism it uses for new nodes. However, it also passes a special "master" flag. This flag signifies to the XNet controller that this "node" should receive capabilities for any new nodes created in the same project, as well as a capability to the special "broker" object that is described in Chapter 5, which is the basis of inter-tenant collaboration. This master workflow-agent is the bridge that connects the concepts of "project" and "user" into the capability world. Additionally, in very simple setups, it can be used to manage all network access policy without having to orchestrate a complex policy setup on multiple nodes, or add capability-aware code to legacy applications.

## 7.2 Controller

As mentioned, the XNet controller is a standard OpenFlow controller that implements the XNet capability model. The controller written in C and based on the OpenMUL OpenFlow application framework. It consists of three main parts: `libcap`, `libobj`, and the wire protocol implementation. `libcap` is a capability library that implements the most basic features of an object capability system. It implements the concept of "capability pointers" contained in "capability spaces" and provides mechanisms to map system pointers to capability pointers, retrieve the system pointers from capability pointers, as well as perform primitive operations like grant, delete, and revoke on capabilities. `libcap` is responsible for implementing the important Capability Derivation Tree (CDT) data structure that tracks the lineage of a capability, and enables objects like Membranes and the `revoke` primitive to work. On top of this framework, the XNet controller adds `libobj`. `libobj` implements the objects described in Chapter 5. These object's implementations directly call the `libcap` API to realize their specifications. Additionally, `libobj` object-specific callbacks to implement special functionality when certain `libcap` methods are invoked like "grant". These callbacks are used to add meta-information to capabilities (like their wrap state), or, in the case of flows, realize the operations in the actual system.

A primary job of the XNet controller is to implement the policy described by the current distribution of flow capabilities in the network. To achieve this, when a flow capability is first granted to a node, the (unidirectional) connectivity is reified into an OpenFlow rule and pushed onto the first-hop switch of the receiving node. By default, the XNet controller implements a "deny all" policy, keeping with the "principal of least authority"; traffic is only permitted to flow after the node receives the appropriate flow capability. Since a single node may hold multiple capabilities to a single flow object, the XNet controller uses reference counting to ensure that a flow is pushed onto the network for only the first capability to a given flow a node receives. When a node loses a flow capability (via `delete`, `revoke`, etc.) the reference count for the flow object is decreased. When the count reaches zero, the flow is deleted from the first-hop switch and the node loses its access. We would like to restate that all capability-related operations happen purely in the control plane. The enforcement of the policy is done in the dataplane, and does not involve the XNet controller being "in the loop".

Finally, the controller implements an interface that allows for capability-enabled nodes in the network to execute capability operations on the controller. As described in Section 7.1, OpenStack informs the controller of the (switch, port) a node is located on. On startup, the XNet controller installs a rule that matches packets with a special capability-protocol ethertype, sending such capabilities to the controller. When a node sends a packet with the correct ethertype, the packet is sent to the controller who parses the message, looks up the node object associated with the (switch, port) the packet was received on, and then attempts to execute the operation. The capability operation is performed as if it was being performed by the actual node object, so only capabilities that are valid for that object are valid in the request. Since the (switch, port) to node binding is unforgeable, this ensures that nodes are only allowed to execute legal capability operations using this mechanism. That is to say, they are only allowed to execute the methods of the objects they have capabilities for. If the operation the node executes returns any results (for example, a capability to a newly created object), the controller constructs a new response packet and directly injects it onto the sender first-hop switch. This ensures that such responses are only sent to the correct sender. However, since capabilities are bound to the sending (switch, port), there is little an attacker could do with such information.

## 7.3 Wire Protocol

The wire-level protocol implemented by the XNet OpenFlow controller is based on Google protocol buffers [4]. It is a request/response-based protocol. Each capability request packet has an "operation" flag that corresponds to one of the primitive operations described in Chapter 5, such as `create` or `invoke`. Capabilities are exchanged as 64-bit numbers. The protocol utilizes the ethernet frame directly and therefore does not benefit from any of the additional features of protocols like TCP. To support certain asynchronous messages, the protocol supports retransmission, but does not support fragmentation and assumes that all messages can be contained in an ethernet MTU. In practice, this is not an issue, as all of our messages are significantly smaller than an MTU.

## 7.4 Client Library

To allow cloud tenants to easily interact with the capability system, we have also implemented an API that abstracts the wire protocol into a simple object-oriented interface that matches the model. The client library utilizes dummy objects that store the capability for the "real" object internally. When the client invokes a method on such a dummy object, the library creates a capability protocol request and sends it to the controller, marshalling the response into the appropriate return object.

In addition to these basic operations, the client library implements some helper utilities for commonly performed routines (like receiving and then resetting a collection of nodes), and the "Application as a Service" protocol described earlier in Chapter 6.

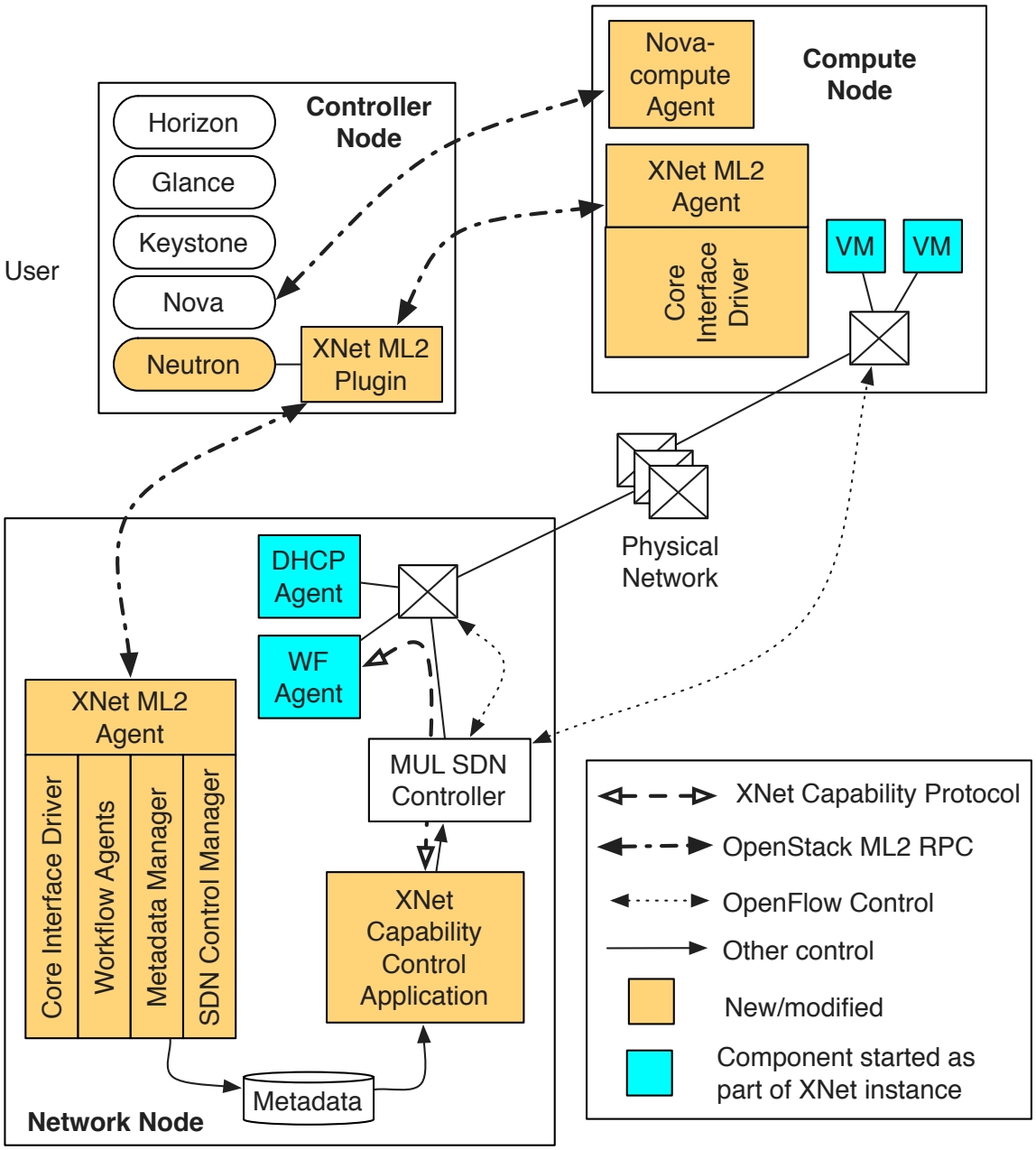


**Figure 7.1**. XNet implementation

# CHAPTER 8

# SECURING GALAXY

Modern bioscience, especially in the area of genetics, heavily relies on the ability to execute a series of statistical and transformational analyses over large (tens to hundreds of gigabytes) datasets. These datasets contain many different types of information such as genetic data, and patient medical records. To aid researchers in performing their analyses, several "scientific workflow" systems have been developed (Taverna [5], Galaxy [2]). Using these scientific workflow systems, researchers can construct "workflows" that consist of a series of "tools" that are piped together in an operation graph. Each tool performs one phase of the analyses the researcher wishes to perform.

These scientific workflows systems are important to researchers for two reasons. First, they allow them to automate complex processes (which may take hours or days to execute) in a user-friendly graphical way. This allows them to more easily construct complex workflows, understand why or how they work (or don't work), and share workflows to allow for easy fully reproducible experiments. Secondly, these workflow systems provide an *infrastructure* that makes it unnecessary for researchers to become system administrators and maintain their own data analysis clusters. These workflow systems are usually provided to researchers as a platform, where a researcher can make an account in a publicly provided system, and then perform their analyses using the infrastructure provided by the service operator. Researchers upload their datasets to this infrastructure, and then use these datasets as inputs to the analysis workflows. The results of these workflows can then be downloaded from the workflow system, or used as the input to further workflows.

Though using these systems drastically lowers the data management and administration workload for researches, little emphasis is put on data security or privacy. Datasets uploaded by the researcher are stored on a global filesystem that is accessible by any tool used by any researcher. Not to mention that all data are visible to the Galaxy system itself.

If a researcher wishes to use confidential information in a scientific workflow system, their only current option is to host it themselves, again becoming system administrators for their own infrastructure.

To address the issues of privacy and security in scientific workflow applications, we apply XNet to an existing scientific workflow system, Galaxy. We leverage the "Application as a Service" protocol to ensure that researcher data stays private, even while being accessed by arbitrary "tool" scripts. We do this with minimal modifications to Galaxy, and no modifications to the "tool" scripts themselves.

A simple overview of the Galaxy architecture is depicted in Figure 8.1. Users interact with the Galaxy system through a web-based frontend. Through this portal, they can create new workflows, or inspect the results of previous workflow executions. At a high level, in Galaxy, a workflow is a graph of "tools" that are arbitrary scripts. Each tool has an XML description that describes the script to execute, as well as any inputs, how they are to be supplied (via standard-in, as arguments, etc.), and any outputs (files, etc.). It's worth noting that there are three types of nodes in this graph: sources, sinks, and transform nodes. Sources are scripts that take no inputs. These nodes are responsible for fetching the data that will be used in the workflow. Sinks are the output of the workflow; they are nodes with no outputs connected to other tools. What we will call the "transformation" nodes are nodes that take input, and produce output. These tools are typically responsible for the actual analysis. Once a user has described a workflow in the Galaxy frontend, they tell the Galaxy frontend to "execute" it. At this point, the Galaxy frontend forwards the workflow to the Galaxy control node. The control node is responsible for orchestrating the execution of a workflow. It has access to a tool database that contains all of the scripts that can be used in workflows. For each node in the workflow graph, the Galaxy control node selects the appropriate tool script from the tool database, and sends it to a free worker node. Additionally, it sends any required input data to the worker node. This worker node executes the script and then sends the resulting data back to the Galaxy control node, who stores it in the workflow data database. Once every node has been executed, the Galaxy control node gives the frontend the results, so they can be displayed to the user.

In order to make this workflow secure, we assume that the tenant is on the same cloud as the Galaxy instance (control node, and worker nodes). We leverage the XNet

Application as a Service protocol to execute the tools in isolation, storing the data on a node provided by the user, instead of a database managed by the Galaxy control node. In the isolated case, the process is very similar to the above. However, when requesting a workflow execution, the user supplies a token (that can be looked up via the broker) to an RP for a data node controlled by the user. Instead of directly executing the tool on the worker node, Galaxy sends a *capability* to the worker node to the user's data node. The user can then use this node capability to execute the Application as a Service protocol against the service provider that will be providing the tool. To show that our system can implement the same abstraction that Galaxy does, we'll consider the Galaxy control node itself to be the tool provider. However, the Application as a Service protocol does not require this, meaning our system actually supports independent and proprietary 3rd-party tools, which Galaxy does not. Once the user's data node has execute the Application as a Service protocol against the Galaxy control node, the control node can install the appropriate tool script that will be executed against the researcher's data. We call the configured node a "tool node" as it is responsible for executing a single tool. After the protocol is complete, the data node can be sure that the tool node is isolated from Galaxy. The tool node will request the input data it needs from the data node it is now connected too. Once the tool is done executing, it can send the result data back to the user's data node. Once the result data has been received, the user's data node signals the control node, which can disable the user's access to the node by revoking the node capability it sent at the beginning of the process. Since the control node does not posses a NodeLease to the Node, it must be reset before it is used in another execution, and therefore will be wiped. Once this process has been done for every node in the workflow, the results will be on the data node, available for the user to access.

## 8.1 Implementation

To implement this extension, we first wrote a XNet-aware job running system. The workflow required by XNet is not a standard job-control workflow, due to the fact that the operator who "installs" the job and the operator who executes the job are separate, physically partitioned entities. The XNet-aware job running system consists of three pieces: The data-node manager, the cluster manager, and a runner. Jobs are sent to the cluster

manager who coordinates the setup and execution of the job. The data-node is controlled by the owner of the data the job is executing on, i.e., the researcher in the case of Galaxy. When a job needs to be executed, a description of the job is sent to the cluster manager. It specifies: an identifier that can be used to lookup a rendezvous point for the data-node, a specification of the job to run, and a list of the input and output files. The identifier is used to bootstrap communication between the data-node and the cluster manager. The cluster manager sends a node to the data-node that the data-node uses to directly perform the Application as a Service protocol against the cluster manager. Once during the phase of the Application as a Service protocol, where the service provider has access to the resources, the cluster manager installs the runner script, and gives it a specification of the job to run. This specification consists of the job that was originally enqueued at the cluster manager, and the input/output mapping that was provided simultaneously. The runner stores this specification until it is given access to the actual data-node at the end of the Application as a Service protocol. At that point, the runner fetches all input data, executes the job, and uploads any output data. The actual credentials or information required to access the data-store are supplied to the runner after the Application as a Service protocol has been completed. The job is considered to be finished once the runner notifies the data-node manager of job completion. At that point, the data-node manager tells the cluster manager that that node is no longer needed. The data-node's rights can be revoked, and the node can be re-added to a set of available worker nodes. In the case that the data-node is malicious and doesn't want to return the worker node, the cluster-manager can employ a timeout and forcibly reclaim the node once the timer expires.

Secondly, we wrote a Galaxy job runner plugin that utilized this XNet-aware job runner to execute the required jobs. Due to the variety of grid computing systems with which Galaxy interoperates, it has a clean separation between the job runner system and Galaxy as a whole. Galaxy exposes a basic `queue_job` interface that provides the plugin with a specification of the tool that will be run, including the set of input and output files the tool will consume or produce. The XNet Galaxy plugin generates the input and output mapping required by bootstraping off of Galaxy's existing data management system. Instead of storing the actual file contents, the plugin stores pointers to the files. It stores the RendezvousPoint identifier for the data-node that holds the files, and the identifier for

the file itself. For the output files, it infers the RendezvousPoint identifier from the input files, and generates random identifiers for the output files. By pregenerating the output file identifiers, the plugin doesn't have to try and recover the location of the output files; it already knows. To start the workflow, the researcher can use a simple "XNet Import Tool" to ensure that the plugin is aware of the input file location. This tool is configured to just write the identifier to the data file, so it can be used as an input for a job run by the actual job runner plugin. To recover the output of a workflow, the researcher can look at the output data file in Galaxy to see the key of the final output. Using that key, they can look up the result in their data-node's data store.

## 8.2 Limitations and Future Directions

While the XNet-aware Galaxy system provides significant improvements in terms of usability, it does have some limitations. The principal among them is that Galaxy's interactive features can no longer be used with the XNet-aware system. By default, Galaxy's tools can supply postprocessing scripts that add metadata to their outputs. The Galaxy control node can then read this metadata, and present a more interactive frontend interface to the user. For example, using this metadata, the Galaxy system can prepopulate fields for the user to select, or perform automatic conversions when a datatype does not match the input datatype for the tool. This is not possible in the XNet-aware Galaxy because XNet enforces a strict separation of the data node and the Galaxy control node. Any generated metadata files are only accessible to the data node, not the Galaxy control node. This is fundamental to the data isolation. If data from the tool-node were allowed to pass back to the data-node, it would be a vector for data leaks. This problem can be solved in two ways. First, the Galaxy system itself could stop providing interactive features that would be broken by XNet to its users. This is preferable to users who would rather have additional security than additional functionality. Finally, a data-node tool for accessing and displaying the metadata could be created. Such a tool would have access to the metadata, and could allow the user access to its meatadata.

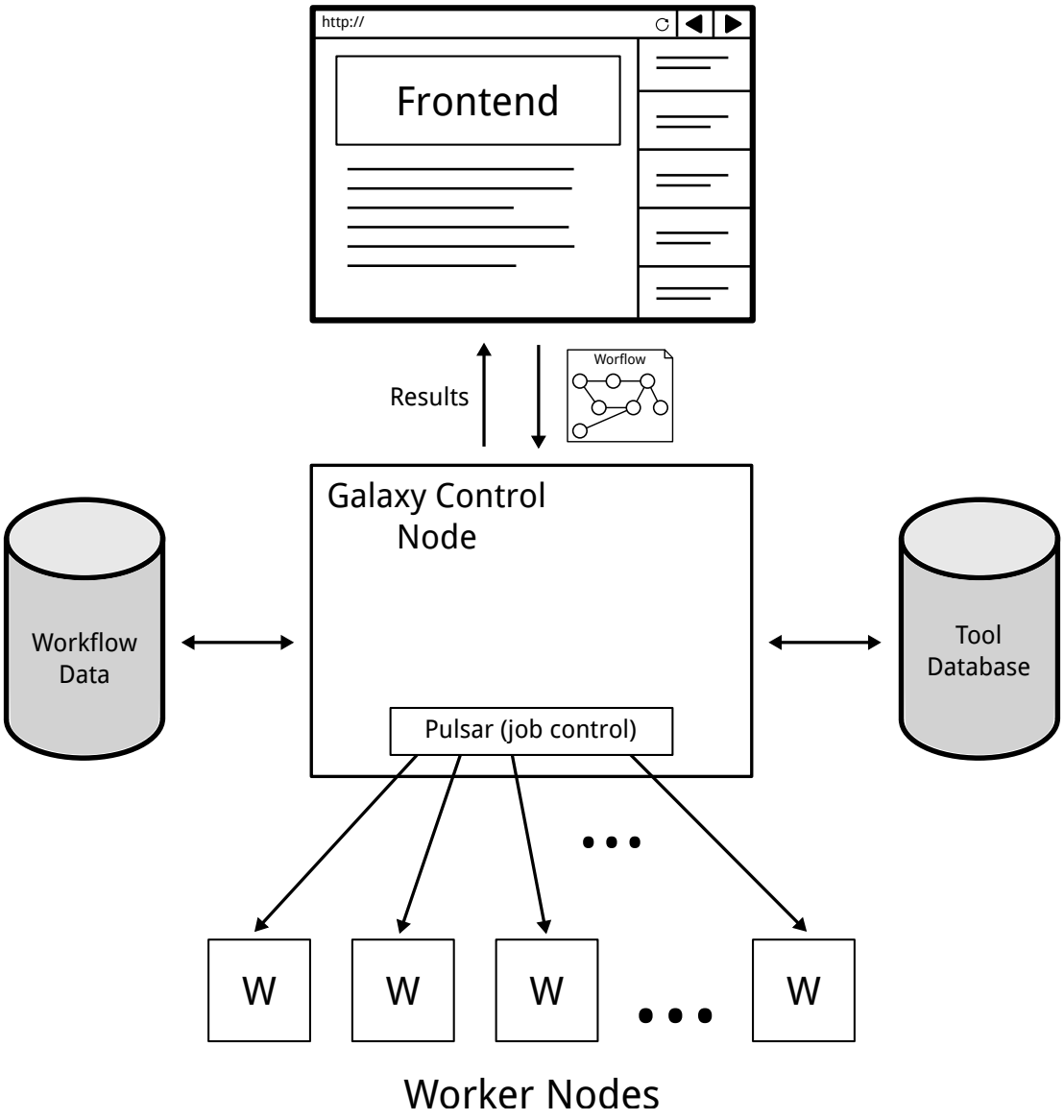


**Figure 8.1**. Architecture of the Galaxy scientific workflow system.

# CHAPTER 9

# EVALUATION

Our evaluation focuses on the performance and scalability of XNet's capability operations in the context of a simple cloud-based, multiparty Application as a Service case study.

We first examine performance of XNet's core capability operations, and show that they are low-cost, and that even the most expensive operations are fast. Second, we discuss performance of the master workflow agents executing capability operations and building communication paths, relative to execution time of a simple Hadoop job, to show that the total time to create a XNet network configuration is reasonable. Third, we evaluate the scalability of the XNet SDN controller by running multiple concurrent AaaS tenant pair experiments, to show that our controller can scale within a multitenant cloud. Finally, we evaluate the XNet-extended Galaxy system described in Chapter 8, showing that it adds additional security and functionality compared to the state of the art.

## 9.1 Application as a Service Software Case Study

We built two master workflow agents, a realization of the "Application as a Service" collaborative XNet protocol described in Chapter 6, each of which is owned and run by a different OpenStack tenant. The first master workflow agent (the "user WFA") receives a list of node capabilities to VMs that were allocated by a user tenant and attached to the XNet network. The second master workflow agent (the "service WFA"), running in a different tenant, registers with the XNet broker object to provide the "Hadoop" configuration service. The user WFA looks up the Hadoop service using the broker object, and using the rendezvous point object capability it receives, sends capabilities to its VMs to the service WFA. Once received by the service WFA, the WFA resets these nodes to clear their capabilities, creates an all-pairs set of flow capabilities, giving each VM the ability to talk to every other (Hadoop requires that all nodes communicate), and installs and configures

Hadoop. Finally, the service WFA sends the grant capability associated with the Hadoop master node back across the membrane to the user WFA. The user WFA then "clears" the membrane, revoking the service WFA's capabilities to the VMs it just configured. Finally, the user WFA runs the Hadoop `wordcount` job on an input dictionary file that is sized according to the number of slave nodes (so that each slave could process a 128 MB file).

### 9.1.1 Experiment Infrastructure

We conducted our experiments on an OpenStack cluster configured with XNet. The cluster contains 26 machines: one node functions as the OpenStack "controller" (authentication and API services); another node as the "network manager" (runs network-wide services such as DHCP, DNS, the OpenStack metadata proxy—and XNet services such as master workflow agents); and compute nodes that host VMs. Each machine is a Dell PowerEdge R430 with two 2.4 GHz 8-core E5-2630 processors, 64 GB RAM, and one 200 GB SSD, running Ubuntu 15.10, Linux kernel 4.2.0-27, OpenStack "Liberty", and Open vSwitch 2.4.0. Each node is connected to a 10 Gb Ethernet LAN (the XNet physical data plane). The network manager and compute nodes each have a single Open vSwitch bridge (containing the physical Ethernet device) controlled by the XNet controller.

### 9.1.2 Test Setup

We ran the master workflow agents on new VMs a total of 60 times: 15 trials each with 50, 100, 150, and 200 slaves. Each set of 15 trials operated on an identical input file (approximately 6.4, 12.8, 19.2, and 25.6 GB, respectively).

## 9.2 Capability Operation Benchmarks

In this section, we examine the overhead of various capability protocol messages. These operations fall into two categories: "Soft" operations that only affect the state of objects on the controller, and "Hard" operations that can affect the network itself (i.e., that cause flow add or removal). Table 9.1 shows the range of observed execution times for Soft operations (as a group) and select Hard operations that are important to the function of the Application as a Service protocol. Most hard operations take only a few hundred microseconds. In the worst case, clearing a membrane involves deleting hundreds of flow capabilities that all manipulate the state of the network, yet it still takes only hundreds of

milliseconds.

The operations `create(Flow)`, `Grant.grant` are the primary vectors for flow capabilities. Since network state is updated when a new flow capability is received, the timings for these two operations represent the expected cost of altering the network connectivity of XNet nodes. Extrapolating from these operation timings, we can see that our network manipulations take only 100-200$\mu$s on average.

The complex `Node.reset` and `Membrane.clear` operations re-isolate a node and destroy a membrane, respectively. `Membrane.clear` is XNet's most costly operation. They are expensive because they may make many changes to the underlying network state. For example, when a membrane is destroyed, flow capabilities on the wrong "side" of the membrane will be deleted; this is the primary motivation for membranes. Figure 9.1 shows a CDF of the time to execute a membrane clear in each experiment. The cost of `Membrane.clear` increases proportional to the large numbers of wrapped capabilities in the larger experiments. In our largest experiment, `Membrane.clear` took no longer than $\approx$ 600ms. Since `Membrane.clear` is only invoked a few times in a protocol to re-isolate exposed nodes, it is unlikely to be a major bottleneck.

When `Node.reset` is invoked to re-isolate a node, all flow capabilities pointing "to" the node must be revoked. Depending on the number of principals that own a given flow to a node that is being reset, this operation can become expensive. A CDF of the execution time of `Node.reset` for each experiment is shown in Figure 9.2. The cost of `Node.reset` is mostly invariant on the number of nodes, since most nodes have the same number of incoming flows when reset. Only the data up to the 99th percentile is show in the graph for clarity of description. The maximum time taken by a `Node.reset` operation was 1.3ms.

## 9.3 Workflow Agent and Hadoop Performance

Here we analyze the "macro" performance of the master workflow agents executing the AaaS protocol. We show the overhead of groups of the costly capability operations, compared to the time spent configuring and running Hadoop. (We do *not* show the "soft" capability operations in these tables, since they are low-cost and not major contributors to total WFA times.)

Table 9.2 lists time spent in key phases in the user WFA, while Table 9.3 lists time spent

in the service WFA.

The user WFA receives capabilities to nodes in its tenant from the controller ("recv-nodes"); it uses a 30 s timeout to detect when the controller has finished sending. We create the service WFA prior to VMs; thus, the "membrane-recv" operation in the service WFA is lengthy (the time includes VM creation). During"membr-wait-recv", the user WFA waits for the service WFA to send the capability back through the membrane, signaling that it has completed Hadoop setup ("hadoop-setup"); and then clears the membrane to revoke capabilities from the service WFA. The user WFA loads data into HDFS and runs a Hadoop job ("hadoop-job-run").

## 9.4 Multiple Simultaneous AaaS Workflow Agents

In this section, we evaluate the scalability of both capability operations and master workflow agent performance by running multiple, concurrent AaaS master workflow agent pairs. In this example, we do not configure or run Hadoop, so the execution of the master workflow agents consists only of capability operations, as well as the time required to boot the VMs. We do this by design to force each master workflow agent pairs' capability operations to operate nearly simultaneously, to encourage parallelism and lock contention at XNet's controller—to allow us to analyze XNet's scalability.

### 9.4.1 Test Setup

We ran an AaaS WFA pair for each tenant, and we increased the number of concurrent tenants. For each of 2, 3, and 4 concurrent tenants, we ran 5 trials, each with 50 and 100 slave nodes. This test does not run Hadoop so we set per-slave RAM to 2 GB and 1 VCPU to achieve greater packing. We do not use 150- and 200-slave tests in this experiment for several reasons. For instance, our tuned Neutron configuration produced errors on large parallel VM creates; this prohibitively increased the test runtime. However, the number of slaves is much less important than the number of competing tenants, which is our focus in this test.

### 9.4.2 Capability Operation Benchmarks

Table 9.4 shows the timings of two capability operations, `Grant.grant` and `Membrane.clear`, depending on the numbers of nodes and parallel AaaS executions (i.e., 2 instances means

4 master workflow applications running). As shown in the table, the execution time for a given capability operation scales similarly to the single instance case (Table 9.1); there is no significant slowdown from running multiple tenants in parallel.

## 9.5 Evaluating Galaxy Extensions

The extensions to the Galaxy scientific workflow system (described in Chapter 8) through the addition of a XNet-aware job running system provides a functional example of how XNet can be used to augment the security of existing systems with relatively minimal overhead. To evaluate these extensions, we will provide a security analysis, showing that the system adds significant security over the state of the art, and a functional analysis to show that a system like Galaxy can incorporate XNet with minimal effort.

### 9.5.1 Security Analysis

To describe how the XNet system augments the security of Galaxy, we must first describe Galaxy's current threat model. Galaxy's threat model is a classic full-trust model. The users must trust the Galaxy workflow system with its data. Both the Galaxy application and the tools it exposes have full read/write access to all data on the system. The Galaxy administrator controls the set of tools, so it can be assumed that these tools are at least as trustworthy as the Galaxy administrator. Instead of isolating users of the system directly, users are expected to run isolated galaxy instances that they themselves administer.

In the XNet augmented Galaxy workflow system, the trust between the user and the Galaxy system is broken. We assume that Galaxy may be directly malicious, or that the tools installed in the Galaxy instance are malicious. We also utilize the Application as a Service protocol, and therefore rely on its security assumptions. Namely, we assume that The XNet model is implemented correctly as described. For our analysis, we will be concerned with three possible actors: the user, a tool, or the Galaxy system itself. The primary goal of the malicious tools or Galaxy systems are to compromise user data. The user's primary goal is to unfairly use Galaxy resources. For simplicity of description, the XNet job runner will be considered a component of the Galaxy system.

First, note that the Galaxy system never has access to the data itself, or even a connec-

tion to the data that is not mediated by the user. The data node only exposes the data to the runner node once the membrane has been revoked and all access to the Galaxy system has been destroyed. Since the Galaxy system must reset the node before establishing a network connection to it, all state will be wiped, ensuring that results from previous job executions are not leaked. Since the Galaxy system never has access to the data, it is clearly not possible for it to compromise the user's data.

The currently executing tool does have temporary access to the data, but it isolated using the Application as a Service protocol. The protocol guarantees that the tool cannot access the data until all of its external access (to the Galaxy system, or to the outside world) is cut off. Even if the tool and the Galaxy system collude, they cannot re-establish a connection after the membrane is cleared. As long as the system for retrieving and uploading data only allows nondestructive writes, the tool cannot tamper with data that already exists on the data node. Additionally, the node may attempt to store data in such a way that it can be retrieved on a later invocation. Since the node is reset with each invocation, no state persists between invocations, and data leaking cannot occur through this mechanism. The tool may attempt to execute a Denial of Service attack against the node, but that is outside of the scope of this threat model.

Clearly, as long as the XNet system functions properly, we can defend against malicious tools and malicious Galaxy system operators. Additionally, since all resources can always be reclaimed by their owners (using a revoke and reset), it is impossible for the user to unfairly hold Galaxy resources.

### 9.5.2 Functional Analysis

Converting Galaxy to use a XNet job runner involved limited modification of the actual Galaxy system itself, though it did disable a number of Galaxy features that could not exist given that the data are not present. The Galaxy system exposed a standard job-queuing interface that meshed well with a XNet enabled job runner. Storing XNet aware pointers to user data, instead of the data itself, disabled interactive features of Galaxy. For example, for certain tools, Galaxy will present a list of options based on the input dataset. Since the XNet datasets were not valid, it was not able to infer that any options were valid. This is fundamental to the data isolation. A system where the data must be separated from the

user interface cannot expose data-dependent behaviour to the user without breaking the isolation of the system. Galaxy is still able to execute most of its operations, and can still use these tools in a noninteractive "workflow-based" mode.

**Table 9.1**. Capability operation timings (min. to 99th perc. in $\mu$s).

| Operation | Number Slave Nodes | | | |
|---|---|---|---|---|
| | 50 | 100 | 150 | 200 |
| Soft | 0-1120 | 0-1160 | 0-1270 | 0-1090 |
| create(Flow) | 20-650 | 50-290 | 30-290 | 40-300 |
| Grant.grant | 120-460 | 100-450 | 100-440 | 70-460 |
| Node.reset | 170-720 | 160-680 | 150-674 | 130-720 |
| Membrane.clear | 52880-72050 | 158340-202030 | 319140-364730 | 505650-594150 |

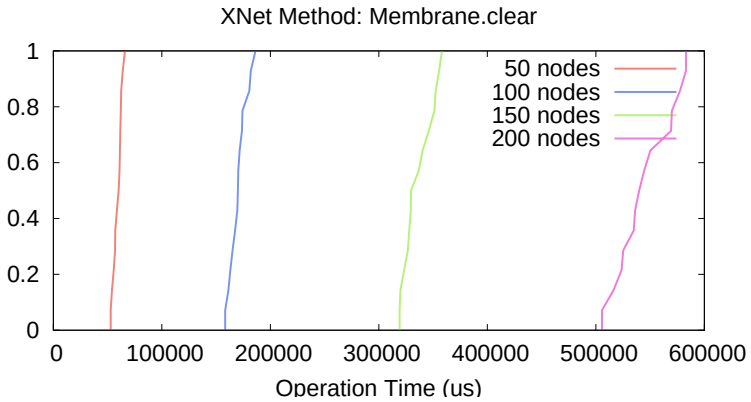


**Figure 9.1**. Membrane clear

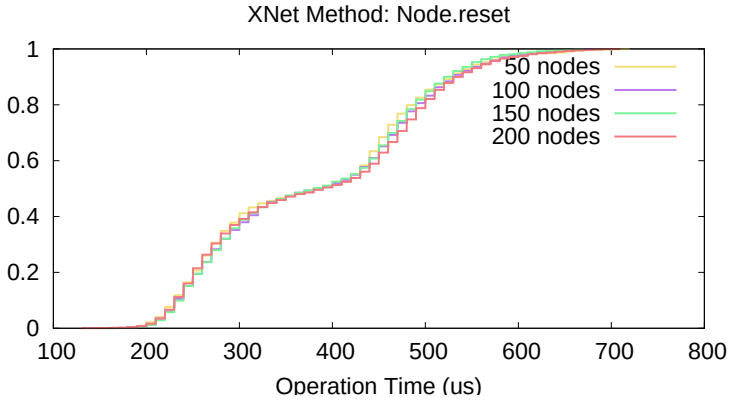


**Figure 9.2**. Node reset (up to 99th percentile)

**Table 9.2**. User WFA time (sec.) in selected functional blocks, and full execution time (columns do not sum total).

| Operation | Number Slave Nodes | | | |
|---|---|---|---|---|
| | 50 | 100 | 150 | 200 |
| recv-nodes | 36.5012 | 42.8794 | 49.5650 | 56.0228 |
| send-nodes | 1.1590 | 2.1302 | 3.0669 | 4.2292 |
| membr-wait-recv | 71.8752 | 122.3947 | 265.3045 | 260.5325 |
| membrane-clear | 0.1005 | 0.2109 | 0.3806 | 0.5902 |
| hadoop-job-run | 151.2563 | 246.9126 | 325.0591 | 432.6685 |
| *full WFA time* | 261.7379 | 419.2941 | 654.1214 | 766.4997 |

**Table 9.3**. Service WFA time (sec.) in selected functional blocks, and full execution time (not sum total).

| **Operation** | **Number Slave Nodes** | | | |
|---|---|---|---|---|
| | 50 | 100 | 150 | 200 |
| membrane-recv | 95.0663 | 143.7468 | 192.3154 | 254.1871 |
| recv-nodes | 8.3839 | 16.2502 | 24.0745 | 32.1975 |
| allpairs | 8.7366 | 33.1489 | 74.0023 | 128.9374 |
| hadoop-setup | 56.0935 | 75.4767 | 170.7995 | 104.2655 |
| *full WFA time* | 168.6481 | 268.8255 | 461.6599 | 519.5675 |

**Table 9.4**. Capability operation timings with parallel Application as a Service (minimum to 99th percentile in $\mu$s).

| **Operation** | **2 instances** | | **3 instances** | | **4 instances** | |
|---|---|---|---|---|---|---|
| | 50 | 100 | 50 | 100 | 50 | 100 |
| Grant.grant | 100-430 | 110-440 | 90-450 | 70-450 | 60-430 | 50-460 |
| Membrane | 38890- | 162880- | 52090- | 141380- | 40040- | 144600- |
| .clear | 67920 | 206510 | 69480 | 201110 | 69020 | 209840 |

# CHAPTER 10

# RELATED WORK

XNet builds on a large body of related work from a wide variety of areas. XNet is primarily a network access control system for clouds, and therefore must be contrasted against existing cloud access control systems. Traditional cloud access control systems use RBAC or RBAC-like [6, 19] models for cloud API-access, and traditional network control systems like Security Groups and VLANs [1, 20, 21] for actual network control. The systems only allow for limited inter-tenant cooperation, and policies are difficult to compose directly from the VLAN and Security Group rules. Additional semantic information is required.

Beyond the cloud domain, there is a significant body of work on network access control where there exists a single global administrator [7,8]. While these works motivate the need for a more dynamic access control policy, they are not directly applicable. Such systems tend to assume the existence of a single, omnipotent administrator that sets the policy for the entire network. While this approach may address the concerns of a single enterprise, it does not generalize to cloud networks.

There is additional previous work that focuses on allowing multiple parties to collaboratively manage an SDN fabric [11, 27]; later extensions [22, 23, 28] show that this work is vulnerable to malicious users that install conflicting rules into the network. Because XNet constructs the network access control definitions such that there cannot be conflicting policy, it is not affected by this issue.

Most directly, XNet builds upon the large body of work related to capability systems. Capabilities themselves were originally formulated by Dennis van Horn [9], and have been used as the primary access control system in operating systems [10, 12, 18, 25, 26] and programming languages [13, 15]. Miller et al. [16] and Watson et al. [30] argue that capability models can be used to construct practical security systems.

XNet is heavily influenced by these prior works, principally in how they affected the design of the system. XNet primarily builds on the design principles used in operating systems. For example, XNet's RendezvousPoints are very similar to seL4 [17], and XNet provides a limited set of objects with defined implementations. The "membrane" objects described in [15] are the basis for XNet's. However, our model is significantly different as our system has a fixed number of objects, so dynamic facets/caretakers are not possible. Therefore, we provide a different model that attempts to achieve the same isolation goals with a single object implementation.

# CHAPTER 11

# CONCLUSION AND FUTURE WORK

Modern cloud access control systems are not capable of handling the requirements of modern cloud workflows. These access control systems require rigid isolation between tenants that doesn't allow for the flexible and contextual trust models of modern infrastructure services. By augmenting a cloud system with a capability model, tenants can more dynamically express their polices. The dynamic and contextual policies even allow for tenant-to-tenant services that are impossible on existing infrastructure, while still maintaining by-default the strict isolation model of the cloud. Through our prototype implementation of the XNet system on OpenStack, we show that this functionality can be provided with little overhead. Further, by augmenting the Galaxy scientific workflow system with a XNet enabled job-runner, we showed that meaningful security could be added to existing systems with little implementation cost. Therefore, we can conclude that augmenting current cloud access control systems with capabilities enhances functionality and security of real-world applications, with low run-time costs.

While XNet is a significant improvement over the state of the art in cloud access control systems, it still has room for improvement. The security systems described in this document are bound to a single cloud. To collaborate, all parties have to be resident on the same cloud. This can be an overly limiting restriction. For example, many modern clouds have multiple sites that allow services providers to provide their services in a globally distributed manner. Further, different cloud architectures present different features, like faster hardware, or different pricing structures. Currently, the XNet architecture assumes centralized control over the cloud system. To support multiregion clouds, or clouds with different operators, XNet could be extended with multidomain support. By allowing capabilities to operate across cloud domains, XNet protocols could allow cloud tenants to collaborate beyond the boundaries of their own cloud. Each tenant would be free to

choose their own cloud operator and associated set of features, without losing the ability to use services provided by remote tenants. If the XNet system can scale beyond the environment described in this thesis, it could serve as a platform for security in future cloud architectures, thereby bringing the advanced collaboration features of XNet to all cloud users.

# REFERENCES


[1] Amazon EC2 Security Groups for Linux Instances. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html.

[2] Galaxy. https://galaxyproject.org/.

[3] Neutron/ML2. https://wiki.openstack.org/wiki/Neutron/ML2.

[4] Protocol Buffers. https://developers.google.com/protocol-buffers.

[5] Taverna. https://taverna.incubator.apache.org/.

[6] Amazon Web Services. AWS Identity and Access Management (IAM). https://aws.amazon.com/iam/.

[7] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. Ethane: Taking control of the enterprise. *Comput. Commun. Rev.* (2007).

[8] Casado, M., Garfinkel, T., Akella, A., Freedman, M. J., Boneh, D., McKeown, N., and Shenker, S. Sane: A protection architecture for enterprise networks. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (Berkeley, CA, USA, 2006), USENIX-SS'06, USENIX Association.

[9] Dennis, J., and Van Horn, E. Programming semantics for multiprogrammed computations. *Communications of the ACM* (1966).

[10] Hardy, N. KeyKOS architecture. *ACM SIGOPS Operating Systems Review* (1985).

[11] Jin, X., Gossels, J., Rexford, J., and Walker, D. Covisor: A compositional hypervisor for software-defined networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015), pp. 87–101.

[12] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. sel4: Formal verification of an OS kernel. In *SOSP* (2009).

[13] Mettler, A., Wagner, D., and Close, T. Joe-E: A security-oriented subset of Java. In *NDSS* (2010).

[14] Microapps. MoonMail. https://aws.amazon.com/marketplace/seller-profile/ref=dtl_pcp_sold_by?ie=UTF8&id=9054b407-f800-4765-9c08-1a5a0acfe30a.

[15] Miller, M. *Robust composition: Towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006.

[16] Miller, M. S., Yee, K.-P., and Shapiro, J. Capability myths demolished. Tech. rep., 2003.

[17] Minsky, N. H. Selective and locally controlled transport of privileges. *ACM Trans. Program. Lang. Syst.* (1984).

[18] Neumann, P., Boyer, R., Feiertag, R., Levitt, K., and Robinson, L. *A provably secure operating system: The system, its applications, and proofs*. SRI International, 1980.

[19] OpenStack. Horizon Policy Enforcement (RBAC: Role Based Access Control). `http://docs.openstack.org/developer/horizon/topics/policy.html`.

[20] OpenStack. Neutron/SecurityGroups. `https://wiki.openstack.org/wiki/Neutron/SecurityGroups`.

[21] OpenStack. OpenStack Networking Guide. `http://docs.openstack.org/newton/networking-guide/`.

[22] Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., and Gu, G. A security enforcement kernel for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks* (2012), ACM, pp. 121–126.

[23] Porras, P. A., Cheung, S., Fong, M. W., Skinner, K., and Yegneswaran, V. Securing the software defined network control layer.

[24] RenderStreet. AWS Marketplace: Professional render farm for Blender and V-Ray. `https://aws.amazon.com/marketplace/pp/B00NMRQ7JG?qid=1474321294605&sr=0-6&ref_=srh_res_product_title`.

[25] Shapiro, J., Northup, E., Doerrie, M., Sridhar, S., Walfield, N., and Brinkmann, M. Coyotos microkernel specification. *The EROS Group, LLC, 0.5 edition* (2007).

[26] Shapiro, J. S., Smith, J. M., and Farber, D. J. EROS: A fast capability system. In *SOSP* (1999).

[27] Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G. Flowvisor: A network virtualization layer.

[28] Shin, S., Porras, P. A., Yegneswaran, V., Fong, M. W., Gu, G., and Tyson, M. Fresco: Modular composable security services for software-defined networks. In *NDSS* (2013).

[29] Stripe, I. Payments For Developers. `https://aws.amazon.com/marketplace/pp/B0087P1DBU?qid=1474323393076&sr=0-2&ref_=srh_res_product_title`.

[30] Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: Practical capabilities for UNIX. In *USENIX Security* (2010).