CapNet: Security and Least Authority in a Capability-Enabled Cloud

Anton Burtsev
aburtsev@uci.edu
University of California, Irvine
Irvine, CA, USA

David Johnson
johnsond@cs.utah.edu
University of Utah
Salt Lake City, UT, USA

Josh Kunz
jkz@cs.utah.edu
University of Utah
Salt Lake City, UT, USA

Eric Eide
eeide@cs.utah.edu
University of Utah
Salt Lake City, UT, USA

Jacobus Van der Merwe
kobus@cs.utah.edu
University of Utah
Salt Lake City, UT, USA

ABSTRACT

We present CapNet, a capability-based network architecture designed to enable least authority and secure collaboration in the cloud. CapNet allows fine-grained management of rights, recursive delegation, hierarchical policies, and least privilege. To enable secure collaboration, CapNet extends a classical capability model with support for decentralized authority. We implement CapNet in the substrate of a software-defined network, integrate it with the OpenStack cloud, and develop protocols enabling secure multiparty collaboration.

CCS CONCEPTS

• **Networks → Security protocols**; *Programmable networks*; *Cloud computing*; • **Security and privacy** → *Access control*;

KEYWORDS

capabilities, capability-enabled network, OpenFlow, OpenStack

1 INTRODUCTION

Despite numerous advances in cloud security, modern clouds remain vulnerable to targeted attacks [10]. Traditionally, cloud security concentrates on perimeter protection. Network firewalls and advanced intrusion-detection systems are designed to keep attacks outside of the cloud network. Today, however, well-sponsored, coordinated, targeted attacks continue to penetrate the cloud perimeter [20, 51]. Once inside, an attacker can leverage an excessive network authority of the cloud network, enabling the discovery and exploitation of other subsystems. In a modern cloud, the cloud network is the main attack amplifier, turning the initial compromise of a single account or a network host into a platform for launching a broad, cloud-wide attack.

SoCC '17, September 25–27, 2017, Santa Clara, CA, USA

The fundamental reason is that the cloud network is built on legacy network-isolation primitives: VLANs, ACLs, and security groups [1, 37, 40]. Taking their roots in the context of enterprise networks, these mechanisms are designed to isolate coarse-grained, largely static networks controlled by centralized administrators. Clouds, on the other hand, embrace dynamic allocation of fine-grained resources, extensive use of third-party services, and collaboration among tenants. Existing network security mechanisms provide no support to minimize the authority of cloud workloads in the face of composable services and collaborating parties. For example, it is not possible for a tenant to safely isolate a subset of his or her nodes and pass them to a third-party service provider to perform setup and configuration of the service, e.g., a distributed file system. Instead, the deployment of a third-party service requires updates to tenant-wide security rules and can break the security of the entire system [23, 24]. Similarly, having no control over the consumer's network, a third-party service provider that deploys its service as a collection of virtual machines inside a consumer's network is required to trust the consumer's environment, i.e., trust that the service will not be attacked [54].

We suggest a new network architecture, CapNet, aimed at implementing *least authority* and *secure collaboration* in the cloud network. At its core, CapNet is an object capability system that represents the resources of a traditional network as a graph of objects that have unforgeable pointers (or *capabilities*) to other objects. In CapNet, objects represent two kinds of resources: (1) resources of the cloud network itself (hosts, network flows, etc.) and (2) primitives that control mutation of the object graph. Capabilities in CapNet allow principals to perform operations on objects: e.g., a capability to a "flow object" allows packets to be sent along the flow, and a capability to a "node object" can control a virtual or physical device in the cloud. Principals have no authority beyond capabilities: all network operations are accessible only through capability invocations.

By controlling the capability graph, CapNet enables powerful security constructs in the cloud network. (1) **Least authority.** The key principle enabling security in CapNet is the ability to construct small, isolated cloud subsystems that operate on a minimal number of isolated resources. By minimizing the authority of individual subsystems, CapNet guarantees that even if a part of a computation is compromised, the possible effect of the attack is minimal and limited to a set of objects reachable through capabilities. (2) **Mutual isolation.** CapNet develops mechanisms enabling mutual isolation of principals irrespective of the ownership of the network nodes and their past. In CapNet, the providers and consumers of third-party cloud services can interact without requiring trust. For example, even if a service is configured by a third-party provider, the consumer of the service has guarantees that the service is completely isolated from the provider after deployment. On the other hand, the provider can protect its proprietary service from the consumer by isolating the service from the consumer itself, inside the consumer's network. (3) **Decentralized access control.** CapNet develops primitives that enable decentralized access control [32] in the cloud network. Cloud workloads that involve multiple parties combining their sensitive datasets for joint processing (e.g., [7, 13, 14]) require isolation for each dataset and the ability to reason about information flow across the joint computation. CapNet allows mistrusting tenants to assemble tightly controlled cloud networks in a decentralized manner, i.e., without trusting each other, yet enforcing their own network-wide isolation and information-flow properties.

We make the following contributions. First, we design an object capability access-control model for the cloud network. While enabling radically new security capabilities, CapNet retains backward compatibility with existing network stacks and requires no modification of the hosts. We implement CapNet with an SDN network fabric and demonstrate how CapNet can be integrated with the OpenStack cloud platform. Second, we extend classical object capability models [29] with support for mutual isolation and decentralized authority. We apply three capability primitives—*reset*, *membranes*, and *sealer/unsealers*—to control mutation of the capability graph. Based on these primitives, CapNet enables mistrusting cloud tenants to develop decentralized capability protocols that enable constructing general network topologies and complex cloud workflows with guarantees of isolation and information flow. Finally, we develop protocols of secure collaboration between untrusted parties in a cloud environment.

CapNet is open-source software and is available for download at https://gitlab.flux.utah.edu/tcloud/capnet. It can be test-driven via a CloudLab [43] profile (https://www.cloudlab.us/p/TCloud/OpenStack-Capnet) that automatically instantiates a private OpenStack cloud for the user, preconfigured with CapNet.

2 BACKGROUND AND RELATED WORK

The industry-standard AWS and OpenStack clouds implement isolation and access control at two conceptual levels. First, a role-based access control (RBAC) model [2, 38] is used to control the boundary of cloud-service APIs that provide access to core cloud functionality, e.g., creating virtual machines, configuring and accessing storage, configuring cloud networks, etc. RBAC allows tenants to control how intra-tenant principals—cloud users and VMs—access cloud APIs within the slice of a tenant's resources. Second, to isolate cloud applications within the network, modern clouds provide tenants with classical network-isolation primitives: VLANs, ACLs, and security groups [1, 37, 40]. Security groups [37], which have become a de facto standard, are responsible for providing fine-grained isolation in the cloud network. CapNet aims to replace security groups as the network access-control mechanism. We argue that security groups fail to provide a consistent framework for manipulating the security state of the system, e.g., splitting a group into isolated subgroups, delegating connectivity to a subset of group nodes, revoking previously granted resources, etc. The lack of delegation in security groups results in excessive network authority and limits the ability of mistrusting tenants to collaborate. Finally, changes to the security-group configuration are always global (and hence, require system-wide reasoning that is prone to error [23, 24]). In CapNet, capabilities enable local and compositional changes, support fine-grained delegation, and allow recursive revocation of any authority in the network.

OAuth 2 [18] and similar authorization systems [6] implement a capability-like protocol that allows cloud services to grant their users certain access to other services (i.e., implement delegation). In contrast to CapNet, these systems rely on cryptographic primitives to protect the credentials of each principal. One limitation is that, based on cryptography, they can implement only the simplest capability protocols: delegation [18] and attenuated delegation (delegation of the subset of rights) [6]. Second, OAuth operates at the level of cloud applications: i.e., the application is responsible for checking the cryptographic token. Hence, OAuth leaves the cloud network open and vulnerable to attacks. Packets from compromised hosts can reach other hosts on the network and exploit vulnerabilities in the network protocols [35] and other parts of the system reachable from the network [21].

The original work on software defined networking (SDN) was motivated by the goal of providing control over enterprise networks [8, 9]. While achieving a global view of the network, and hence a possibility of enforcing fine-grained network-wide security policies [8, 9], these approaches still largely view the network as an entity with a single administrative root. In practice, enterprise and cloud networks consist of numerous, mutually mistrusting workflows and tenants that are governed by many principals with conflicting security goals. Network slicing [25, 47] addresses the problem of isolation among mistrusting principals by virtualizing a single SDN network, but it still fails to provide mechanisms allowing decentralized authority and security in the face of collaboration among mistrusting parties. Numerous systems try to extend SDN with access-control primitives aimed at enabling some forms of controlled cross-tenant communication and collaboration. The FortNOX Enforcement Kernel [42] and its extensions Fresco [48] and SE-Floodlight [41] point out the problem that multiple (possibly malicious) SDN controller applications can install conflicting flow rules that violate network-wide security policies. In contrast, CapNet builds on the principles of object capability systems that allow the composition of security rules by design, and hence make it impossible to install rules that contradict an initial policy.

Initially formulated by Dennis and Van Horn [12], capability systems became a popular mechanism for constructing secure, least-privilege environments in the areas of operating systems [19, 27, 33, 45, 46] and programming languages [28, 29]. The development of the object capability model [29] revived capabilities as a viable abstraction for constructing least-privilege security environments [27, 52]. Miller et al. [30], and later Watson et al. [52], provide a good discussion of the advantages of the capability model for constructing practical security environments. CapNet's model builds on the design principles of capability-based microkernels [15, 19, 34, 46] and object capability languages [29]. Similar to seL4 [15], CapNet (1) uses send and receive [31] to model the grant operation and (2) implements capability spaces and derivation trees. CapNet builds on the work of Miller [29] on capability design patterns that extend the capability model with principles for constructing secure systems in the face of mutually mistrusting parties and diverse security requirements. CapNet further extends the object capability model with design patterns that provide support for decentralized authority through the novel use of membranes, reset, and sealer-unsealers.

Capability concepts have been applied to networking in previous work, but in the context of denial-of-service (DoS) attacks. For example, capability tokens have been suggested as a means to prevent DoS attacks [3]. Tokens implement a capability-like authorization mechanism allowing communication in the network, but are limited to expressing only this simple property. More recent work also proposed to deal with unwanted traffic through capability mechanisms, but instead of issuing capabilities, proposed to use dynamically changing IP addresses *as* the capabilities [49]. In CapNet, no network communication is possible unless explicitly allowed by a capability. As such, our work is related to earlier "off by default" approaches [5].

3 THREAT MODEL

CapNet is an access-control infrastructure that controls connectivity in a cloud network. A CapNet configuration is a security policy that identifies a set of principals (i.e., physical or virtual hosts), the communication paths between those principals, and the ways in which the policy can be modified by those principals. The goal of an attacker is to violate the policy defined by a CapNet configuration by (1) causing unauthorized communication to occur, (2) preventing authorized communication, or (3) causing an unauthorized change to the policy.[1]

We make the following assumptions about threats to a CapNet implementation. We assume that the cloud provider is trustworthy and the provider's network infrastructure (switches, routers) is secure and free of vulnerabilities. Further, we assume that the cloud tenants and providers of third-party cloud services are untrusted. We assume that an attacker controls an endpoint device attached to the network managed by CapNet, i.e., a physical or virtual host. The only actions available to the attacker are to send and receive network packets, including packets to or from nodes known to CapNet, nodes unknown to CapNet, and the CapNet controller. While an attacker may try to forge packets, as it fully controls its host, it cannot hide its location. Because the network infrastructure is trustworthy, CapNet can reliably identify the ⟨switch, port⟩ pair at which the attacker is attached to the network.

4 CAPNET ARCHITECTURE

Logical level. CapNet represents a traditional cloud environment as a graph of objects connected with edges that represent rights, or *capabilities* (Figure 1). An *object* represents an entity within

[1]An extended low-level threat model would also consider quality-of-service attacks, including attacks that exhaust resources such as SDN flow-table rules. We do not consider QoS attacks in this paper.

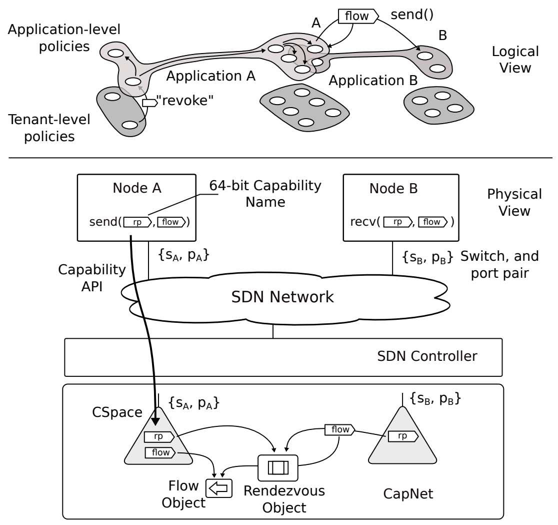


Figure 1: Internals of the CapNet capability system

CapNet, such as a specific virtual or physical network device or a communication endpoint. Objects "live" within the CapNet controller, and thus they are like objects in an object-oriented programming language. Objects have types, state, and methods; methods accept parameters and return results. A *capability* is a reference to a particular object. Capabilities are unforgeable: one cannot "make up" a capability to an object. The only way to receive capabilities is through the traditional rules of object capability systems [29]: initial conditions, parenthood (creating new nodes), endowment (receiving from own creator), and introduction (receiving through communication channels). The users of a CapNet network use capabilities to invoke operations on objects: in this way, users communicate with each other and implement the security policies they require. In a capability system, the only way to access a resource is to invoke a capability that enables access to the object that represents that resource in the capability system. Network communication, host management, and the exchange of rights are mediated and controlled by the rules of the CapNet capability system.

Physical level. To ensure isolation, CapNet operates in the context of a cloud environment where hosts (physical and virtual) are connected to a software defined network (SDN) (Figure 1). The CapNet SDN controller prevents all communication until explicitly allowed. CapNet is responsible for ensuring consistency between the logical and physical state of the system. When a state of the capability system requires a change in the physical network, the CapNet controller uses SDN mechanisms to implement the change, e.g., push a new flow-table entry into the network.

Hosts invoke capabilities via the CapNet capability API (Figure 1). The API maps onto a simple, reliable network protocol that conveys API invocations to the CapNet controller. CapNet manages the serialization of capability operations. Each capability protocol message contains a capability that identifies an object, an operation to perform on the object, and zero or more arguments. Most capabilities are held by Nodes, objects that represent a virtual or physical host, and that are the only "active" objects in our system. Software

Table 1: CapNet object types and their methods

Node	
	`grant reset()`
	`object create(object_type type, specification spec = ())`
Flow	
RendezvousPoint	
	`void send(cap c, string msg)`
	`(cap, string) recv(int timeout)`
Grant	
	`any invoke(cap c, method m, args args)`
Membrane	
	`cap wrap(cap c)`
	`void clear()`
SealerUnsealer	
	`cap seal(cap c)`
	`cap unseal(cap c)`
Operations on capabilities	
	`cap mint(cap c, specification spec = ())`
	`void revoke(cap c)`
	`void delete(cap c)`

on the host refers to capabilities by their local names—64-bit capability identifiers that have no special meaning outside of the host (Figure 1). For each host, the CapNet capability system maintains a *capability space* (or a *CSpace*), a fast data structure that resolves local capability identifiers into capability references that contain specific rights to a specific object in the capability system. CapNet relies on the SDN substrate to provide a unique ⟨switch, port⟩ mapping for all network communication. This mapping is used to identify the correct *CSpace*, thus ensuring that capability identifiers cannot be forged.

4.1 Object Types

CapNet uses a small number of object types, summarized in Table 1, to define the configuration of a network.

A **Node** represents a network-attached virtual or physical device—i.e., a virtual machine instance. Nodes are the only objects allowed to create other objects and initiate capability operations. When the software running on a device interacts with CapNet, e.g., by making API calls, it acts "on behalf of" the Node object that is currently associated with the device. In other words, the Node is the *principal* associated with the software running on the device. A newly created Node object is "born with" one capability to a special RendezvousPoint, *rp0*, that connects it to its creator and allows it to perform the boot protocol. The Node may acquire or lose capabilities over time as the software interacts with the CapNet API. Nodes are allowed to create other objects by invoking **create**(Type), where *Type* is one of the object types. Nodes are allowed to create Flows to themselves, RendezvousPoints, Membranes, and SealerUnsealers (discussed below). Grant objects are only created as the result of the **reset**() operation (see Section 4.3).

A **Flow** represents a unidirectional communication channel: the ability to send packets to a particular network endpoint. In CapNet, the endpoints that send and receive packets are Nodes, so Flows are strongly tied to Nodes. If *node* is a capability to a Node, node.**create**(Flow, flow_spec) creates a new Flow that allows communication to *node*. The optional *flow_spec* adds restrictions to the flow, e.g., specifying TCP, UDP, or source or destination port. The method returns a capability to the Flow object.

If a Node holds a capability to a Flow, then the device that the Node represents can send packets along that Flow to the Flow's destination. CapNet does not model packet transmission and receipt explicitly. Instead, nodes send and receive packets using traditional network stacks, and the CapNet controller enforces the rules about who can talk to whom through SDN flow-table entries. Sending a packet to an IP address is an *implicit method invocation* on a Flow object whose target Node is associated with that IP address. For the transmission to succeed, the source Node must hold a capability to a suitable Flow. A Node that holds a capability f to a Flow can invoke **mint**(f, spec) to create a "subflow." This makes it possible for one Node to delegate a more restrictive subflow to another Node.

A **RendezvousPoint** allows Nodes to exchange capabilities, e.g., exchange a pair of Flow capabilities to establish connectivity. A RendezvousPoint implements a queue in which each element carries a capability and a string. If a Node holds a capability *rp* to a RendezvousPoint, it can invoke rp.**send**(cap) to insert *cap* into the RendezvousPoint's queue. The RendezvousPoint now holds a copy of the capability that was passed to **send**(); the sender does not lose its capability. A Node can invoke rp.**recv**() to retrieve the element at the head of the RendezvousPoint's queue. After an element is returned by **recv**(), the RendezvousPoint deletes its copy of the returned capability. The **recv**() call blocks until an element can be returned; an optional *timeout* argument forces **recv**() to wait for a defined period and return an error if no element is available.

Capability operations. CapNet defines three operations on capabilities themselves (Table 1). **Mint** returns a new capability that is a copy of the given capability, perhaps with a restricted subset of rights defined by the *spec* argument. **Delete** removes a capability from the collection that the caller holds. **Revoke** allows the caller to recursively delete a previously-granted capability, removing the authority conferred by the capability to any principal the owner granted it to, and so on. The owner's capability is not destroyed. Similarly to seL4 [27], we implement recursive revocation with capability derivation trees (CDTs). A capability to a new object becomes a root of a CDT. When a capability is sent through a RendezvousPoint to another Node, the capability that is inserted in the receiver's *CSpace* is added to the CDT as a child of the capability of the sender. Similarly, every minted capability is derived from its parent. When a capability is revoked, CapNet walks the CDT and removes all children of the revoked capability linked by the CDT from every *CSpace*.

4.2 Support for Unmodified Hosts

A **Grant** object is designed to include unmodified legacy nodes with no knowledge of the capability system in a capability-enabled network. Capability-aware network nodes use Grant objects to operate on behalf of passive legacy nodes, i.e., create objects on their behalf, enable network connections, etc. Every Node has a Grant object associated with it. The Grant implements the **invoke**(cap, m, args) method that allows the owner of a Grant capability to invoke a method *m* of the object referenced by *cap* with arguments *args* as if the caller was the Node associated with the Grant object. CapNet provides other operations for manipulating Nodes through the grant interface as well. Grant.**grant**(cap c) allows the caller to insert a capability into the *CSpace* of the Node pointed to by the Grant object.

Similarly, Grant.**take**(capability_id cap_id) inserts a specific capability, identified by *cap_id* in the *CSpace* of the Grant object's Node, into the *CSpace* of the caller. Finally, Grant.**create**(object_type **type**) creates a new object, inserts a capability to that object into the *CSpace* of the Grant's Node, and conveniently also "takes" this capability to the caller's *CSpace*. For example, **grant.create**(Flow) creates a Flow to the Node pointed to by the *grant* capability. Similarly, the following code connects two Nodes *A* and *B* pointed to by the *grantA* and *grantB* capabilities.

```
flowA = grantA.create(Flow)
flowB = grantB.create(Flow)
grantA.grant(flowB)
grantA.grant(flowA)
```

While Grant is primarily intended to support legacy nodes, it also implements a convenient programming paradigm that allows one to configure multiple network hosts in a "passive" manner. In this style, Nodes themselves do not invoke capability operations; instead, all configuration is done by an "active" node that has grant capabilities. This allows us to implement general security protocols as functions that operate on grants provided as arguments.

4.3 Decentralized Authority and Collaboration

Three mechanisms extend CapNet's capability model to enable decentralized authority and secure collaboration among mistrusting cloud tenants: a reset operation, membranes, and sealer-unsealers.

The **Reset** operation (Node.**reset**()) allows the owner of a Node capability to reset the node to a clean, isolated state irrespective of its prior state and ownership. Reset effectively re-isolates the Node in the capability system by "cleaning up" all capabilities throughout the system that enable any authority on the node other than ownership. For example, if a service provider receives a capability to a Node as a part of the request to configure its service, the service provider can reset the node, ensuring that the consumer cannot connect to the node unless explicitly authorized by the provider. The consumer still "owns" the node after reset, as reset does not affect the consumer's Node capability. Thus, it can claim the node back by revoking all capabilities it previously granted to the provider, and then resetting the node back into the clean state, but does not have any other authority on the node.

The state and authority of the Node consists of three parts: (1) capabilities that the Node holds to the rest of the system that might enable future communication with other Nodes, and exchange of rights; (2) other Nodes in the network might hold capabilities allowing them to establish future connections to the node using the Flow capabilities to the Node, and capabilities to the Node's Grant interface; and (3) the execution state of the Node's associated device, which may contain sensitive information. Reset cleans up a Node's state through the following three steps: (1) To remove all authority from the Node, **reset**() invokes **delete**() on each capability held by the Node. This ensures that the Node can no longer access any objects in the system. (2) CapNet cleans up all resources in the network that might provide any authority on the node. To track this authority, when a node is created or reset, a new Grant object is created and a capability to it is inserted in a private *CSpace* controlled exclusively by the CapNet controller. This capability acts as the root of the CDT for every Flow capability created for the reset node. When the network state needs to be reset, a **revoke**() operation is performed on the Grant capability, which recursively deletes all flows to the reset node. (3) Finally, **reset**() reboots the host associated with the Node, wipes its persistent storage, and copies a pristine OS onto it. Reset leaves the Node with the capabilities to itself and to a fresh *rp0*. Reset returns a capability to a fresh Grant object, which allows the caller to reconnect with the node by invoking **take**().

```
grnt = node.reset()
rp0 = grnt.take(RP0)
```

Membranes. CapNet provides recursive revocation of capabilities with the **revoke**() operation. Revocation alone, however, does not guarantee isolation of capability graphs. For example, if two otherwise isolated graphs are allowed to access the same RendezvousPoint, *rpA*, they can immediately exchange a capability to another RendezvousPoint, *rpB*. Hence, even if *rpA* is revoked, the graphs stay connected via *rpB*, which allows an uncontrolled exchange of rights, i.e., capabilities to flows, grants, nodes, etc.

To provide transitive isolation of capability graphs, CapNet relies on membranes [29, 50]. A Membrane object guarantees that all capabilities exchanged through it will be revoked when the membrane is "cleared." One can think of a Membrane as setting up an elastic "wall" in the capability graph. To define the behavior of membranes, we rely on labels: every capability may have an attached set of labels. Exchanging a capability through a membrane causes the received capability to be labeled with that membrane ("wrapped"), or have that membrane's label removed ("unwrapped"), as described below. For example, let *m* be a capability to a Membrane object *M*. One passes a capability through this membrane by calling m.**wrap**(). If *rp* is a capability to a RendezvousPoint, w_rp = m.**wrap**(rp) sets *w_rp* to a new capability that is a copy of *rp*, except that it is internally labeled with *M*, i.e., $w\_rp_{\{M\}}$. This encodes that it is "on the other side of *M*." Moreover, all capabilities that are sent or received by accessing the RendezvousPoint through the capability $w\_rp_{\{M\}}$ will also "pass through" *M*. In the example below, the Node capability *node* is sent through the unlabeled capability *rp*, but received via *w_rp*.

```
rp.send(node)
w_node = w_rp.recv()
```

Because *w_rp* is labeled with *M*, the capability returned by w_rp.**recv**() is also labeled with *M*, indicating that it has passed through *M*: i.e., $w\_node_{\{M\}}$. If this capability is again sent through *w_rp*, it passes back through *M*, and the received capability will *not* be labeled with *M*. Consider the following code:

```
w_rp.send(w_node)
node = w_rp.recv()
```

On the first line, because $w\_rp_{\{M\}}$ is labeled with *M*, CapNet causes the RendezvousPoint to receive a capability that is exactly like $w\_node_{\{M\}}$, except that its *M* label has been removed. The capability that is inserted into the RendezvousPoint thus has no label. When that capability is received via $w\_rp_{\{M\}}$ on the second line, the received capability is again wrapped: $node_{\{M\}}$.

Membranes are composable with the rest of the capability system. The rules for labeling ensure that all "derived capabilities"—capabilities that are obtained from capability operations that take a labeled capability as an argument—are also labeled. For example, if the holder of a labeled $node_{\{M\}}$ capability resets the node (**grant** = node.**reset**()), the newly created *grant* will follow the rules of labeling and will be labeled as $grant_{\{M\}}$.

Intuitively, the addition of the label records that the capability has "crossed through *M*," and its removal records that it has crossed back. When a labeled capability passes through the same membrane again, returning to its original side, it is unlabeled. A capability is usable until any of the Membranes in its label set is cleared. When a Membrane is cleared, CapNet revokes all capabilities that are labeled with that Membrane, hence re-isolating the graphs. This allows a powerful security template in which one principal, e.g., a third-party service provider, can configure the system of another principal, e.g., a cloud service consumer, through the membrane, but then be completely isolated when the membrane is cleared (Section 5.1). Finally, membranes compose with themselves. CapNet ensures the capability that crosses multiple membranes is annotated correctly, e.g., $cap_{\{M_1,M_2,M_3\}} = M_1.$**wrap**$(M_2.$**wrap**$(M_3.$**wrap**$(cap_{\{\}})))$

SealerUnsealers. SealerUnsealer objects provide a way to protect a capability while it is being passed through a chain of untrusted principals. A SealerUnsealer is an object with two methods: **seal**() and **unseal**(). The *seal* method seals a capability passed as an argument, i.e., $sealed\_c_{\{SU\}}$ = su.**seal**(c). The returned capability is labeled as being "sealed", i.e., $sealed\_c_{\{SU\}}$, changing its behavior with respect to other capability operations. It is still possible to mint, revoke, delete, send, and receive a sealed capability; these operations return another sealed capability. However, a sealed capability does not give authority to use or change (i.e., invoke methods upon) the object it points to. For instance, a sealed capability to a Flow does not allow packets to be sent along the Flow. The *unseal* method takes a capability sealed by the same SealerUnsealer object and returns the original unsealed capability, i.e., $c_{\{\}}$ = su.**unseal**($sealed\_c_{\{SU\}}$). The **unseal**() operation fails if the label does not match the SealerUnsealer object.

SealerUnsealer objects have a special feature: CapNet membranes allow capabilities to SealerUnsealer objects to pass through unlabeled. Thus, when a membrane is cleared, the capabilities to SealerUnsealer objects are not revoked. In the example below, the capabilities *rp* (a capability to a RendezvousPoint) and *su* (a capability to a SealerUnsealer), are wrapped by the membrane *m*. After the membrane is cleared via m.**clear**(), the wrapped capability *w_rp* is invalid, as it crossed the membrane and is revoked, but the capability *w_su* remains valid.

```
w_rp = m.wrap(rp)
w_su = m.wrap(su)
m.clear()
w_rp.send(cap) # error, w_rp is revoked
s_cap = w_su.seal(cap) # ok, w_su is valid
```

CapNet allows capabilities to SealerUnsealers to cross membranes unrecorded, because SealerUnsealers themselves do not enable communication nor the exchange of capabilities. They can only seal and unseal other capabilities. Thus, capabilities to SealerUnsealers do not violate isolation of object graphs enforced by membranes. In fact, SealerUnsealer objects allow us to reconnect otherwise isolated graphs in a controlled manner, by protecting capabilities with seals while they are passed through a chain of untrusted parties (Section 5).

5 SECURE COLLABORATION PROTOCOLS

CapNet utilizes the above abstractions—reset, membranes, and sealer-unsealers—to create composable collaboration patterns: general security protocols that enable construction of complex cloud systems assembled by mistrusting tenants.

5.1 Secure Provider

CapNet allows the implementation of a *secure provider* protocol in which the consumer and provider of a third-party service in a cloud are not required to trust each other, can both protect their private data, and the provider can protect her proprietary service implementation. We assume that a cloud tenant has a sensitive data set in a cloud and wishes to analyze it using a third-party cloud service (e.g., a large-scale data-analytics platform like Hadoop [4]). Several security constraints guide the design of this protocol. The tenant wants the service provider to install and configure the service, and requires that (1) during installation, the provider should have potentially unlimited access to the hosts dedicated for the service, but should be isolated from the rest of consumer's cloud environment; and (2) once installation is complete, the guarantee that the provider is completely isolated from the service. The provider, on the other hand, requires (1) full unrestricted access to a subset of the consumer's cloud resources to deploy the service (e.g., copy disk images, run performance tests); and (2) a mechanism to protect the service from the consumer after installation is completed and the consumer starts using the service.

Mutual isolation of principals. The "secure provider" protocol builds on the ability of CapNet to provide mutual isolation of principals with reset and membranes. Reset creates a trusted, isolated execution environment inside a potentially untrusted environment. Hence, when the service provider obtains capabilities to the nodes from the consumer, it can reset the nodes in a clean, isolated state even though the consumer still owns the node.

Membranes, on the other hand, provide a symmetric isolation guarantee for the owner of a node. If the consumer initially shares a capability to the nodes with the service provider through a membrane, despite the fact that the provider is allowed unlimited connectivity to the node, the node will be isolated from the provider after the membrane is destroyed. We illustrate these ideas by developing a complete example of the *secure service provider* protocol: a general protocol that allows deployment of complex cloud services consisting of proprietary software and sensitive state and data sets.

Secure provider protocol. **Step 1. Consumer: membranes, isolating the object graphs.** The consumer initiates the protocol by requesting installation of a service with the secure_provider() function (Listing 1). The function takes two arguments: *provider_rp*, a capability to the rendezvous point connecting the consumer with a third-party service provider, and *nodes*, an array of node capabilities on which the service will be installed. To ensure isolation from the provider, the consumer creates a membrane (*membrane*) and a rendezvous point (*rp*) (Listing 1, lines 2–3). It then wraps the rendezvous point (Listing 1, line 4) and shares it with the provider

```
1   cap secure_provider(cap provider_rp, cap nodes[])
2       membrane = create(Membrane)
3       rp = create(Rp)
4       wrapped_rp = membrane.wrap(rp)
5       provider_rp.send(wrapped_rp)
6       for node in nodes {
7           rp.send(node)
8       }
9       srv_rp = rp.recv()
10      membrane.clear()
11      return srv_rp
```

Listing 1: Secure provider protocol, consumer's code

```
1   void do_secure_provider(cap consumer_rp)
2       wrapped_rp = consumer_rp.recv()
3       while (nodes[i] = wrapped_rp.recv())
4           grants[i] = nodes[i].reset()
5       srv_rp = install_service(grants)
6       wrapped_rp.send(srv_rp)
7       return
8   cap install_service(cap grants[])
9       ...
10      srv_rp = grants[j].create(Rp)
11      return srv_rp
```

Listing 2: Secure provider protocol, provider's code

through a regular rendezvous point *provider_rp* (Listing 1, line 5). The consumer then uses the unwrapped rendezvous point to send capabilities to the nodes that will be used to install the service (Listing 1, lines 6–8). Since node capabilities are sent through the unwrapped capability *rp* on the consumer's side, but received by the provider through the wrapped capability $wrapped\_rp_{\{M\}}$ (where M is the membrane object created by the consumer in Listing 1, line 2), the capability system labels all node capabilities on the provider's side as $node[i]_{\{M\}}$. The consumer waits for the provider to finish the installation by performing a blocking receive on the *rp* rendezvous (Listing 1, line 9).

Step 2. Provider: receiving and isolating the nodes. The provider starts the protocol by waiting on the *consumer_rp* rendezvous point connecting it to the consumer (Listing 2, line 2). It receives a wrapped rendezvous point (*wrapped_rp*) from the consumer and then uses it to receive node capabilities to install the service (Listing 2, lines 3–4). To ensure that the nodes are isolated from the consumer, and are in a clean state, the provider resets each node it receives (Listing 2, line 4). Note that while mediated by the membrane, the provider can freely exchange capabilities with the nodes it received from the consumer, establish connections to its internal services (e.g., image and storage servers), and configure connectivity between the nodes.

Step 3. Provider: service install. The provider continues with installation by invoking a custom function specific to the service (*install_service()*, Listing 2, lines 8–11). The provider creates a rendezvous point (*srv_rp*) that will serve as an interface to the service (e.g., exchanging flow capabilities to establish connectivity with consumer's nodes using the service). After installation finishes, the provider returns this capability to the consumer. Note that while *srv_rp* is created by the provider, it will be labeled as $srv\_rp_{\{M\}}$, where M is the membrane object created by the consumer in Listing 1, line 2. Therefore, when $srv\_rp_{\{M\}}$ is sent back to the consumer through the $wrapped\_rp_{\{M\}}$ capability that is also labeled with M, the consumer receives an unwrapped rendezvous capability $srv\_rp_{\{\}}$ which therefore will remain intact after the *membrane* is destroyed. The same is true for all internal Flows, RendezvousPoints, and other capabilities configured by the provider through the membrane on the nodes it received from the consumer.

Step 4: Consumer: destroying the membrane. After service installation completes, the appliance consumer destroys the membrane (Listing 1, line 10). At this point, all capabilities that carry the M label are revoked and therefore the service is isolated from the consumer. Any connectivity that was created between the nodes and the provider is gone.

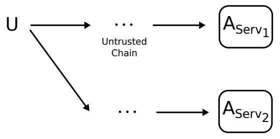
5.2 Trees and General Graphs

Membrane **wrap**() and Node **reset**() are composable operators. They can be invoked recursively among multiple cloud tenants to construct complex services assembled from components provided by each tenant. Specifically, a combination of membranes and reset enables construction of any cloud topology that is a *tree* (Figure 2a). For each parent node in the tree, a tenant asks one or more service providers to deploy their cloud services, assembling an isolated subtree. The tenant then assembles subtrees into a service and returns to its own parent. At each level, the consumer and provider communicate through a membrane; thus, the consumer has a guarantee that each subtree is isolated from the rest of the system, irrespective of any capability operations performed inside the recursive invocations. Moreover, at each level, service providers can deploy their services in isolation by re-isolating the nodes with **reset**().

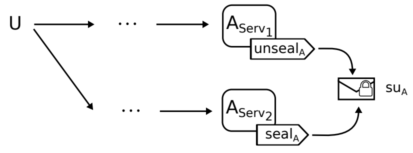
From trees to general graphs. Membranes and reset allow the construction of trees in capability graphs. SealerUnsealer objects extend these mechanisms with a means to securely construct cloud topologies that are general graphs (Figure 2). CapNet relies on the special property of the capabilities to an SealerUnsealer object to pass through membranes unlabeled. Hence, it is possible to construct a tree with a node that has a capability to a SealerUnsealer shared with a principle outside of the tree (in Figure 2b, $unseal_A$ is a capability to a SealerUnsealer, su_A, shared with A). Later, A can leave the capability to the same SealerUnsealer object, su_A, in another part of the tree. The $seal_A$ capability can be used to seal other capabilities (e.g., a capability to a RendezvousPoint, rp_A). The sealed capability $seal_A(rp_A)$ can be sent through the tree to the point where it can be unsealed, thus reconnecting the trees.

5.3 Joint Computation

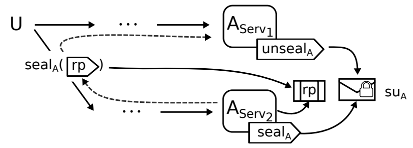
We illustrate the use of SealerUnsealer objects with the "*joint computation*" protocol that allows mistrusting tenants to process federated datasets in a secure manner. In our example, tenants A and B would like to process their combined data sets A_{Data} and B_{Data}, but in such a way that data sets cannot be leaked to either of the tenants (Figure 3). Both tenants require unlimited access to each data set from their compute environments ($A_{Compute}$ and $B_{Compute}$), but would like to control how results of the computation are released to each tenant via validation proxies ($A_{Validation\ Proxy}$ and



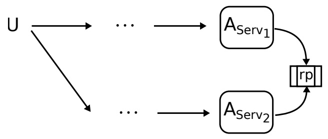
(a) Tenant U uses membranes and reset recursively to construct a general tree of isolated services.



(b) While constructing its services the service provider A leaves the $unseal_A$ capability in one of the services (A_{Serv_1}), and the $seal_A$ capability in another(A_{Serv_2}).



(c) A_{Serv_2} seals the RendezvousPoint capability rp to pass through the system in a protected manner.



(d) A's services A_{Serv_1} and A_{Serv_2} are now connected.

Figure 2: Example of constructing of a general service graph with SealerUnsealer

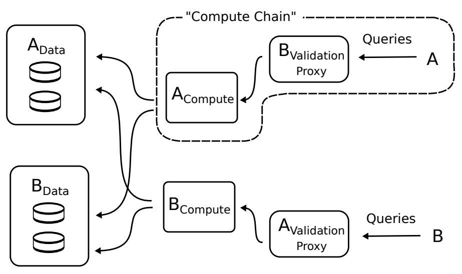
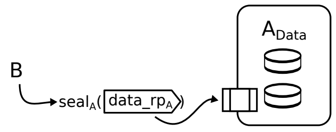


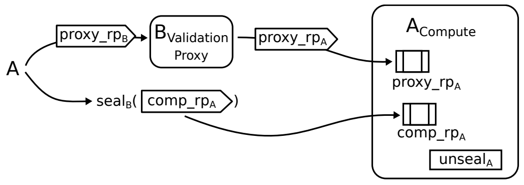
Figure 3: Joint computation

$B_{Validation\ Proxy}$). In other words, the joint computation must be constructed in such a manner that the two parties are guaranteed that: (1) both data and compute components are completely isolated from the outside world; and (2) the only access to compute environments is via validation proxies (Figure 3).

At its core, the protocol consists of three steps. First, the data sets A_{Data} and B_{Data} are deployed by A and B in isolation. Second, "compute chains" for both A and B are constructed as a two-step recursive application of the "secure provider" protocol. Third, "compute chains" are re-connected to the data sets via sealed capabilities.



(a) Step 1: A completed the "secure provider" protocol for B and installed its dataset.



(b) Step 2: B completed the second step of the two-step recursive protocol by installing its proxy and returning to A.

Figure 4: Steps 1 and 2 of the "joint computation" protocol

Step 1. Installing data sets. To deploy its data set in isolation to B, A executes the "secure provider" protocol for B (Figure 4a shows the result of this step). B is guaranteed that the dataset is completely isolated from A. Note that A returns a sealed capability $seal_A(data\_rp_A)$ that can be unsealed by a SealerUnsealer object that is accessible to A through its $unseal_A()$ capability. A plain unsealed capability $data\_rp_A$ would allow B to access A's dataset.

Step 2. Assembling compute chains. To assemble the A compute chain, A first requests B to perform the "secure provider" protocol. B deploys its validation proxy $B_{Validation\ Proxy}$ and recursively requests A to deploy its compute environment $A_{Compute}$. Inside $A_{Compute}$, A leaves a capability $unseal_A$ (Figure 4b) that will allow the isolated compartment of A's compute environment to reconnect to both data sets in Step 3. A returns to B two capabilities (Figure 4b): 1) a capability $proxy\_rp_A$ to a request proxy RendezvousPoint, that will allow establishing connections with the compute cluster, and servicing requests; and 2) a capability to a rendezvous point $comp\_rp_A$ that will be used to reconnect A's compute environment with both A and B's datasets in Step 3.

From inside the "secure provider" protocol initiated by A, B installs its validation proxy that mediates all communication to A's compute environment. B returns to its caller A two capabilities (Figure 4b): 1) a rendezvous capability $proxy\_rp_B$ to allow connections to B's validation proxy; and 2) a sealed capability $seal_B(comp\_rp_A)$. Tenant B seals the second capability to prevent A from obtaining unmediated access to A's compute environment.

Step 3. Connecting compute and data nodes. Finally, to connect both compute environments to each of the datasets, both A and B leave SealerUnsealer capabilities inside A_{Data} and B_{Data}. We illustrate how $A_{Data}$is connected to the $B_{Compute}$ environment (B_{Data} is connected in a similar manner). Because of the result of Step 1, B has access to the sealed capability $seal_A(data\_rp_A)$. B knows that this capability originates from a completely isolated environment (by the guarantees of the secure provider protocol), and can only point to something inside that compartment. B cannot share this capability directly with A, since A will gain access to the A dataset and can establish communication channels to exfiltrate sensitive data from the joint computation later. However, it is safe for B to share with A a sealed capability $seal_B(seal_A(data\_rp_A))$.

This capability can be unsealed only inside B's compute environment, since this is where B left the $unseal_B$ capability. B therefore shares $seal_B(seal_A(data\_rp_A))$ with A. A cannot use this capability directly, but A passes it to B's compute environment. Note that B's computation has the $unseal_B$ capability to remove its seal, but will not be able to remove the $seal_A$ seal. In CapNet, seals commute. It is safe for A to remove $seal_A$ seal from the $seal_B(seal_A(data\_rp_A))$ capability before passing it to $B_{Compute}$, since $B_{Compute}$ has no access to other capabilities protected with $seal_A$. Therefore, A first removes its seal and then shares $seal_B(data\_rp_A)$ with $B_{Compute}$.

6 CAPNET IN OPENSTACK

We implemented the CapNet capability-controlled network and integrated it into the OpenStack cloud platform. Figure 5 depicts our implementation in the context of OpenStack. We expose CapNet as a virtual network type in OpenStack—meaning administrators and tenant users can allocate CapNet networks and ports, attach VMs to them, and use the capability API to create data-plane flows.

6.1 CapNet SDN Capability Controller

The CapNet SDN control application is a multi-threaded C program, built on the OpenMUL SDN controller [36], that manages the flow tables of one or more OpenFlow-enabled switches. The controller maintains all capability state and enforces secure capability invocation and exchange.

Capabilities. CapNet uses `libcap` [22], a general-purpose capability library we created that implements *CSpace*s, CDTs, and the core capability operations (i.e., **mint**(), **revoke**(), and **delete**()) described in Section 4.1. It can be extended with new capability object types and operations on those objects; this is the basis for the CapNet objects (e.g., Node and Flow). The design and implementation of `libcap` is further described in Jacobsen's thesis [22].

Data-plane management. CapNet's controller enforces the network-access policy described by the current set of flow capabilities. CapNet proactively pushes and removes flows to and from its switches, as capabilities are granted and revoked, so the forwarding tables in these switches reflect the current policy. The CapNet controller incurs almost no data-path overhead as its enforcement is done purely via control-plane mechanisms (other than to support non-capability-aware legacy nodes and to emulate broadcast traffic). Node metadata—e.g., MAC and IP addresses, provided by a trusted administrator—is used to restrict flows according to flow capabilities and to prevent spoofing attacks. The CapNet controller does not currently handle controller-switch network partitions; we assume that the cloud infrastructure switch fabric and control plane are reliable. We leave robust CapNet behavior in the face of network partitions and other infrastructure failures as future work.

Capability protocol. Nodes in a CapNet-controlled network interact with the controller using a simple, reliable capability protocol. Capability protocol messages are intercepted using OpenFlow rules and handled in the controller; responses are injected back onto the network. CapNet uses metadata provided by the administrator to associate a $\langle \text{switch}, \text{port} \rangle$ pair with a particular Node object. The controller performs the requested action in the context of the Node object's *CSpace*, ensuring that only legal operations are performed. This snooping-based protocol ensures that devices cannot impersonate each other. The protocol is built atop Google protocol buffers [17]; CapNet provides Python bindings to build capability applications.

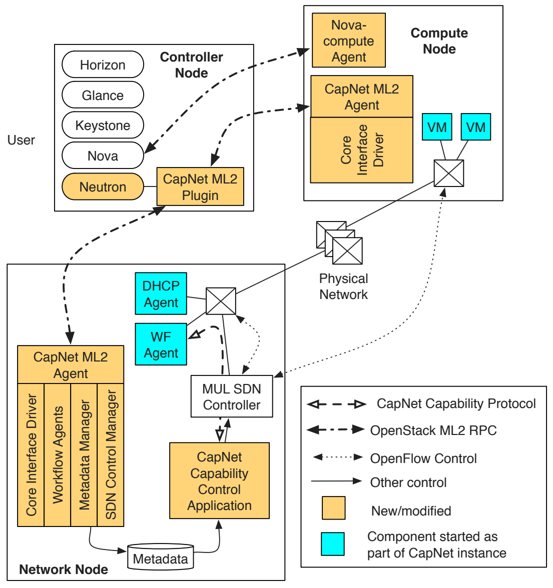


Figure 5: CapNet integration with OpenStack

Performance. The CapNet controller is multi-threaded and safely parallelizes the handling of OpenFlow events and capability protocol messages via fine-grained locking. Multiple hosts might issue conflicting capability operations. To serialize and guarantee the atomicity of capability operations, we rely on the `libcap` library and CapNet controller. All operations that manipulate capabilities acquire locks to the nodes of the CDT and *CSpace* data structures.

Legacy support. To allow unmodified host networking stacks to function, CapNet enables several protocols that rely on layer 2 ambient authority to broadcast in order to locate endpoint information (e.g., ARP, DHCP, etc.). The CapNet controller services ARP requests and preemptively pushes flows to enable DHCP broadcast requests between a DHCP agent and a known server.

6.2 OpenStack Integration

We added CapNet to OpenStack Liberty by exposing it as an OpenStack virtual network type inside Neutron, OpenStack's network service. Figure 5 shows a classic OpenStack deployment. A *controller* node runs user-accessible, high-level RESTful API services that create and manage cloud objects such as VMs and virtual networks. *Compute* nodes host VMs and virtual network ports. A *network* node provides virtual networks and services like DHCP and cloud-metadata services for booting VMs. The service APIs provide authentication, handle cloud resource management at an abstract level, and communicate with distributed agents over internal RPCs to create specific virtual resources (e.g., a CapNet network port).

Neutron plugins implement the details of layer 2 and layer 3 virtual networks and ports when created by administrators or tenants [39]. We wrote a CapNet agent-based driver for the ML2 plugin, making CapNet networks available to cloud tenants.

CapNet networks in OpenStack. The CapNet driver core runs within the Neutron API service on the OpenStack controller node. It interacts via RPC with CapNet driver agents on network and compute nodes to create networks and ports. The CapNet driver configures the CapNet SDN controller, running on the network node, to control the Open vSwitch switches in the infrastructure (typically one on each of the compute and network nodes). Since CapNet provides isolation via capabilities, CapNet virtual networks are marked "shared" in OpenStack, allowing multiple tenants to connect to the same network—and thus collaborate.

CapNet controller management. Coupling between OpenStack and the CapNet network occurs via a Metadata Manager and an SDN Control Manager in the CapNet ML2 Agent on the network node. The SDN Control Manager runs the OpenMUL controller and the CapNet capability control application. The Metadata Manager communicates OpenStack node and port metadata to the CapNet controller, enabling it to enforce flow capabilities. The controller also pushes flows to enable the OpenStack metadata service, through which VMs obtain configuration (e.g., user public keys).

Workflow agents. In CapNet, all nodes connected to ports on CapNet-controlled switches can send capability protocol messages to exchange capabilities. However, nodes running legacy applications may not be capability-aware, or the tenant user may prefer to centrally coordinate all capability-based data-plane paths by writing a program that creates data-plane paths by exchanging capabilities on behalf of the nodes it owns. We call these programs *workflow agents*, and we have added support for them in Neutron. Users create workflow agents (via a simple Neutron API extension) that run a program of their choosing, and may declare that the workflow agent is a *master* for its tenant. Master workflow agents receive node capabilities to all VMs created in that tenant, which allows them to manage capabilities on behalf of their nodes. When a workflow agent is created, the CapNet ML2 agent runs the workflow agent in an isolated Linux network namespace [26], with a network interface, and connects the interface to the specified switch (similarly to how Neutron connects DHCP agents to networks).

Collaboration. To allow OpenStack tenants to collaborate, CapNet provides a broker object that exposes two methods, **register**() and **lookup**(). Workflow agents may register a RendezvousPoint with a service name; other agents may look it up and use it.

6.3 CapNet in Other Clouds

Although we have only integrated CapNet with OpenStack, CapNet could be integrated into other clouds as well. First, CapNet requires full control of an SDN switch fabric to ensure that nodes only send traffic according to flow capabilities, and that capability control-plane messages are securely exchanged. Second, CapNet requires the cloud to provide identifying metadata about nodes plugged into its SDN fabric (e.g., MAC and IP addresses), as well as the tenant that owns them. CapNet uses this metadata to populate the switches with appropriate flow match rules. The cloud trusts CapNet to correctly implement its virtual network semantics (e.g., plug nodes into the correct switch ports). Finally, CapNet must be extended to support network paths that the cloud requires for infrastructure services, such as VM boot-time self-configuration.

7 EVALUATION

7.1 Security Analysis

The threat model for CapNet's implementation was presented in Section 3. Here we describe how our implementation addresses those threats. Recall that the attacker may be located on a device managed by CapNet or on an unknown device.

If the attacker is on an unknown device, there is no corresponding Node in the CapNet controller. Consequently, the attacker cannot hold any capabilities to Flow objects, and there cannot be any Flows with the attacking device as their destination. CapNet creates SDN flow-table entries only for authorized flows, which are explicitly represented by Flow objects and capabilities. A packet that does not belong to any authorized flow is dropped at the SDN switch. Thus, an attacker at an unknown device can neither send nor receive packets.

If the attacker is located on a device that is associated with a Node, it can use the capabilities held by the Node to attempt to violate the current policy. The attacker can attempt to send packets to other nodes or to the controller.

Sending a packet to another Node is an implicit method invocation on a Flow object; it succeeds if and only if the sender has a Flow capability that matches the packet. If CapNet correctly implements the model in Section 4, the attacker is able to send packets only in accordance with the current policy. Sending a packet along a Flow never changes the current policy.

Packets sent to the controller encode method invocations on objects. The attacker cannot invoke methods on objects for which it does not hold a capability. Thus, the attacker cannot change the policy (or send packets, or cause packets not to be delivered) by manipulating those objects. The attacker *may* change the policy by invoking methods on objects for which it holds a capability. Any such change, however, is *authorized* by the current policy, and thus is not disallowed. Given a correct implementation of the CapNet model, the attacker cannot perform authorized actions in order to effect unauthorized policy changes, cause unauthorized packets to be received, or cause authorized packets to be dropped.

Finally, because a flaw in the CapNet implementation could allow an attack to succeed, we examine the size of its trusted computing base (TCB). The core of CapNet is its SDN control application, which totals 10,006 SLOC (C, Python, IDL) as of August 2017. CapNet depends on `libcap` [22]), the capability library we wrote, which totals 2,604 SLOC (C). The CapNet OpenStack plugins total 3,957 SLOC (Python). CapNet also depends on OpenMUL, the SDN controller library we used; OpenStack Liberty (particularly its network and compute components); various C and Python toolchain components; and the Linux kernel and Open vSwitch module (and user space configuration tools). CapNet trusts OpenStack to provide correct network identity and tenant ownership information for VMs; trusts OpenMUL to properly insert flow match rules into Open vSwitch virtual switches; and trusts Open vSwitch to properly forward traffic according to those rules (and would need to trust OpenFlow-enabled switches in a hardware deployment). With the exception of OpenMUL, these components are present in typical OpenStack deployments and are well-tested. In summary, the size of the CapNet components is reasonably small (16,567 SLOC), and do not add significantly to the overall TCB of an OpenStack cloud.

7.2 Performance

We evaluate the performance and scalability of CapNet's capability operations in the context of a simple cloud-based, multi-party "secure provider" case study. In contrast to traditional layer 2 networks, CapNet requires path construction (flow capability grants) before the data plane is usable. Once a path has been constructed after a flow capability grant, the CapNet controller adds no overhead to the data plane—data plane traffic flows exist due to flow capability grants, and CapNet does not inspect the data plane. Thus, CapNet's capability operations must add minimal overhead to SDN flow insertion and removal operations; and when combined into agents that create typical network data paths, execute quickly relative to the runtime of an application *using* the data paths. We focus our evaluation on capability operations, rather than typical SDN metrics (OpenMUL has been characterized elsewhere [44]).

We first examine performance of CapNet's core capability operations, showing that they are low-cost, and that even the most expensive operations are fast. Second, we show that the total time to create a CapNet network configuration is reasonable by comparing the execution times of workflow agent data path construction to the times of a simple Hadoop job. Third, we show that the CapNet SDN controller can scale within a multi-tenant cloud by running multiple concurrent AaaS tenant pair experiments.

Secure provider case study. We built two workflow agents, a realization of the "secure provider" collaborative CapNet protocol described in Section 5.1, each of which is owned and run by a different OpenStack tenant. The first workflow agent (the "user WFA") receives a list of node capabilities to VMs that were allocated by a user tenant and attached to the CapNet network. The second agent (the "service WFA"), running in a different tenant, registers with the CapNet service broker to provide the "Hadoop" configuration service. The WFAs then execute the "secure provider" protocol to install Hadoop, revoke the service provider's access, and execute a simple Hadoop job (`wordcount`) on the configured system.

Experiment infrastructure. We conducted our experiments on the CloudLab testbed [43], in an OpenStack cluster configured with CapNet. The cluster contains 26 machines: one node functions as the OpenStack "controller"; another node as the "network manager" (runs network-wide services such as DHCP and CapNet 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 GbE LAN (the CapNet physical data plane). The network manager and compute nodes each have a single Open vSwitch bridge (containing the physical Ethernet device) controlled by CapNet.

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

7.2.1 Capability Operation Benchmarks. Capability operations fall into two categories: "Soft" operations affect only the state of objects on the controller, and "Hard" operations that can affect the network (i.e., that cause flow add or removal). Table 2 shows the range of observed execution times for Soft operations (as a group)

Table 2: Capability Operation Times (min to 99th% in μs)

Operation	Number Worker 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

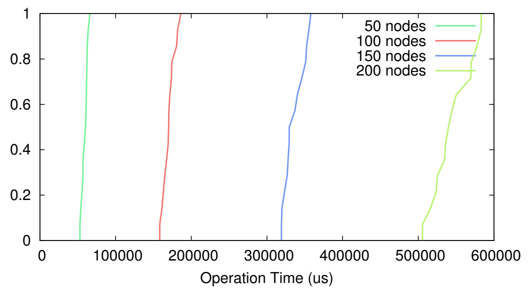


Figure 6: Membrane Clear

and select Hard operations that are important to our case study. 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 network, but 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 operations represent the expected cost of altering the network connectivity of CapNet nodes. Extrapolating from these operation timings, we can see that our network manipulations take only 100-200μs on average.

The complex Node.**reset**() and Membrane.**clear**() operations re-isolate a node and destroy a membrane, respectively. Membrane.**clear**() is CapNet's most costly operation because it may make many network changes. For example, when a membrane is destroyed, flow capabilities on the wrong "side" of the membrane will be deleted. Figure 6 shows a CDF of the time to execute a Membrane.**clear**() in each experiment; its cost increases proportional to the large numbers of wrapped capabilities in larger experiments. In our largest experiment, Membrane.**clear**() took less than $\approx$ 600ms. Since Membrane.**clear**() will be invoked only 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. Its cost depends on the number of principals that own a flow to the node being reset. Figure 7 shows a CDF of Node.**reset**() execution times for each experiment (only data up to the 99th percentile is shown for clarity). The cost of Node.**reset**() is mostly invariant on the number of nodes, since most nodes have the same number of incoming flows. The maximum time taken by Node.**reset**() was 1.3ms.

7.2.2 Workflow Agent and Hadoop Performance. Here we analyze the "macro" performance of the AaaS workflow agents. We show the overhead of groups of costly capability operations, compared to the time spent configuring and running Hadoop. We do *not* show "soft" operations since they are not major contributors to total WFA times. Note that these workflow agents spend significant

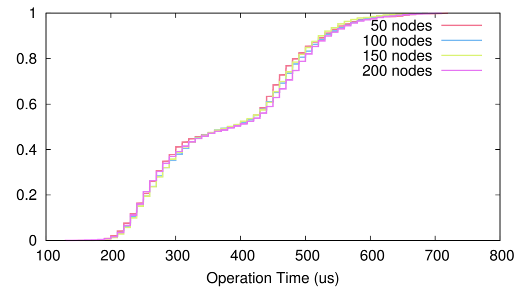


Figure 7: Node Reset (up to 99th percentile)

Table 3: User WFA time in sec. (columns do not sum to total)

Operation	Number Worker 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 4: Service WFA time in sec. (columns do not sum to total)

Operation	Number Worker 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
all-pairs	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

amounts of time waiting for each other's operations to complete—for instance, the user WFA spends significant time simply waiting for the service WFA to set up Hadoop.

Table 3 shows time spent in key phases in the user WFA, while Table 4 shows 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, and this timeout is included in the "recv-nodes" operation time. Immediately after receiving its node capabilities, the user WFA sends its nodes across the membrane to the service WFA ("send-nodes"). After all node capabilities have been sent, it begins waiting for the service WFA to set up Hadoop ("membr-wait-recv"). We create the service WFA before creating the user VMs; thus, the service WFA "membrane-recv" operation is lengthy, because the time the service WFA spends waiting to receive the membrane includes VM creation, as well as the "recv-nodes" operation in the user WFA. The service WFA receives all the node capabilities sent across the membrane ("recv-nodes"), and sets up an all-pairs flow mesh amongst the VMs by granting each VM the ability to send traffic to every other VM. 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").

Table 5: Capability operation times with parallel Application-as-a-Service (min to 99th percentile in μs)

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 .clear	38890-67920	162880-206510	52090-69480	141380-201110	40040-69020	144600-209840

7.2.3 *Multiple Simultaneous AaaS Workflow Agents.*

In this section, we evaluate scalability by running multiple, concurrent AaaS workflow agent pairs. In this experiment, we do not configure or run Hadoop, so the execution of the 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 workflow agent pairs' capability operations to operate nearly simultaneously, to encourage parallelism and lock contention at CapNet's controller—to allow us to analyze CapNet's scalability.

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 worker nodes.This test does not run Hadoop so we set per-worker RAM to 2 GB and 1 VCPU to achieve greater packing. We do not use 150- and 200-worker 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 worker nodes is much less important than the number of competing tenants (workflow agents).

Capability operation benchmarks. Table 5 shows the timings of the Grant.**grant**() and Membrane.**clear**() operations depending on the numbers of nodes and parallel AaaS executions (i.e., 2 instances means 4 workflow applications running). Execution times scale similarly to the single instance case (Table 2); there is no significant slowdown from running multiple tenants in parallel.

8 CONCLUSION

CapNet is a novel network architecture that enables least authority and secure collaboration in a modern cloud. CapNet extends the classical capability model with primitives that enable decentralized authority and the realization of secure cross-tenant collaboration protocols. We implemented CapNet and integrated it with OpenStack. We developed protocols for the secure deployment of cloud services and federated computations. The evaluation of our prototype demonstrates the feasibility of our approach. CapNet is open-source [16] and can be test-driven via a CloudLab [43] profile [11].

ACKNOWLEDGMENTS

We thank our shepherd, Iqbal Mohomed, and the anonymous SoCC reviewers for their valuable comments on this work. We performed our experiments in CloudLab [43], on machines that are part of the Utah Emulab cluster [53]. This material is based upon work supported by the National Science Foundation under Grant Nos. 1547457, 1319076, and 1314945.

REFERENCES

[1] Amazon Web Services, Inc. 2017. Amazon EC2 Security Groups for Linux Instances. (2017). http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html

[2] Amazon Web Services, Inc. 2017. AWS Identity and Access Management (IAM). (2017). https://aws.amazon.com/iam/

[3] Tom Anderson, Timothy Roscoe, and David Wetherall. 2004. Preventing Internet Denial-of-service with Capabilities. *SIGCOMM Comput. Commun. Rev.* 34, 1 (Jan. 2004), 39–44. https://doi.org/10.1145/972374.972382

[4] Apache Software Foundation. 2017. Welcome to Apachae Hadoop! (2017). https://hadoop.apache.org/

[5] Hitesh Ballani, Yatin Chawathe, Sylvia Ratnasamy, Timothy Roscoe, and Scott Shenker. 2005. Off by Default!. In *Proc. HotNets.* http://conferences.sigcomm.org/hotnets/2005/papers/ballani.pdf

[6] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. 2014. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Proc. NDSS.* https://doi.org/10.14722/ndss.2014.23212

[7] Axel Bruns. 2013. Faster than the speed of print: Reconciling 'big data' social media analysis and academic scholarship. *First Monday* 18, 10 (Oct. 2013). https://doi.org/10.5210/fm.v18i10.4879

[8] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. 2007. Ethane: Taking Control of the Enterprise. In *Proc. SIGCOMM.* 1–12. https://doi.org/10.1145/1282380.1282382

[9] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. 2006. SANE: A Protection Architecture for Enterprise Networks. In *Proc. USENIX Security.* 137–151. https://www.usenix.org/legacy/event/sec06/tech/casado.html

[10] Cloud Security Alliance, Ryan Ko, and Stephen S. G. Lee. 2013. Cloud Computing Vulnerability Incidents: A Statistical Overview. (March 2013). https://cloudsecurityalliance.org/group/cloud-vulnerabilities/#_downloads

[11] David M. Johnson. 2017. OpenStack-Capnet CloudLab Profile. (2017). https://www.cloudlab.us/p/TCloud/OpenStack-Capnet

[12] Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9, 3 (March 1966), 143–155. https://doi.org/10.1145/365230.365252

[13] DIAbetes Genetics Replication And Meta-analysis (DIAGRAM) Consortium, Asian Genetic Epidemiology Network Type 2 Diabetes (AGEN-T2D) Consortium, South Asian Type 2 Diabetes (SAT2D) Consortium, Mexican American Type 2 Diabetes (MAT2D) Consortium, Type-2 Diabetes Genetic Exploration by Next-generation sequencing in multi-Ethnic Samples (T2D-GENES) Consortium, et al. 2014. Genome-wide trans-ancestry meta-analysis provides insight into the genetic architecture of type 2 diabetes susceptibility. *Nature Genetics* 46, 3 (2014), 234–244. https://doi.org/10.1038/ng.2897

[14] Dany Doiron, Paul Burton, Yannick Marcon, Amadou Gaye, Bruce H. R. Wolffenbuttel, Markus Perola, Ronald P. Stolk, Luisa Foco, Cosetta Minelli, Melanie Waldenberger, Rolf Holle, Kirsti Kvaløy, Hans L. Hillege, Anne-Marie Tassé, Vincent Ferretti, and Isabel Fortier. 2013. Data harmonization and federated analysis of population-based studies: the BioSHaRE project. *Emerging Themes in Epidemiology* 10, 1 (2013). https://doi.org/10.1186/1742-7622-10-12

[15] Dhammika Elkaduwe. 2010. *A Principled Approach to Kernel Memory Management.* Ph.D. Dissertation. University of New South Wales. http://ssrg.nicta.com.au/publications/papers/Elkaduwe:phd.pdf

[16] Flux Research Group. 2017. Capnet: An SDN Controller and Tools for Capability-based Networks. (2017). https://gitlab.flux.utah.edu/tcloud/capnet

[17] Google, Inc. 2016. Protocol Buffers. (2016). https://developers.google.com/protocol-buffers

[18] Dick Hardt. 2012. *The OAuth 2.0 Authorization Framework.* RFC 6749. Internet Engineering Task Force (IETF). http://www.rfc-editor.org/rfc/rfc6749.txt

[19] Norman Hardy. 1985. KeyKOS architecture. *SIGOPS Oper. Syst. Rev.* 19, 4 (Oct. 1985), 8–25. https://doi.org/10.1145/858336.858337

[20] Imperva. 2013. How Malware and Targeted Attacks Infiltrate Your Data Center. (2013). http://www.ten-inc.com/presentations/Imperva_How_Malware_and_Targeted_Attacks_Infiltrate_Your_Data_Center.pdf

[21] Intel Corporation. 2017. INTEL–SA–00075: Intel Active Management Technology, Intel Small Business Technology, and Intel Standard Manageability Escalation of Privilege. (May 2017). https://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00075&languageid=en-fr

[22] Charles Jacobsen. 2016. *Lightweight Capability Domains: Toward Decomposing the Linux Kernel.* Master's thesis. University of Utah. http://www.flux.utah.edu/paper/235

[23] Cheng Jin, Abhinav Srivastava, Yu Jin, and Zhi-Li Zhang. 2014. Secgras: Security Group Analysis as a Cloud Service. In *Proc. ICNP.* 215–220. https://doi.org/10.1109/ICNP.2014.42

[24] Cheng Jin, Abhinav Srivastava, and Zhi-Li Zhang. 2016. Understanding Security Group Usage in a Public IaaS Cloud. In *Proc. INFOCOM.* https://doi.org/10.1109/INFOCOM.2016.7524508

[25] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. 2015. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *Proc. NSDI.* 87–101. https://www.usenix.org/node/188955

[26] Michael Kerrisk and Eric W. Biederman. 2017. namespaces - overview of Linux namespaces. (May 2017). http://man7.org/linux/man-pages/man7/namespaces.7.html

[27] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proc. SOSP.* 207–220. https://doi.org/10.1145/1629575.1629596

[28] Adrian Mettler, David Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java. In *Proc. NDSS.* https://www.isoc.org/isoc/conferences/ndss/10/pdf/20.pdf

[29] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. Johns Hopkins University. http://www.erights.org/talks/thesis/markm-thesis.pdf

[30] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. 2003. *Capability Myths Demolished.* Technical Report SRL2003–02. Johns Hopkins University Systems Research Laboratory (SRL). http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf

[31] Naftaly H. Minsky. 1984. Selective and Locally Controlled Transport of Privileges. *ACM Trans. Program. Lang. Syst.* 6, 4 (Oct. 1984), 573–602. https://doi.org/10.1145/1780.1786

[32] Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. In *Proc. SOSP.* 129–142. https://doi.org/10.1145/268998.266669

[33] Peter G. Neumann, Robert S. Boyer, Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson. 1980. *A Provably Secure Operating System: The System, Its Applications, and Proofs.* Computer Science Laboratory Report CSL–116. SRI International. http://www.csl.sri.com/~neumann/psos/psos80.pdf

[34] Mark Nevill. 2012. *An Evaluation of Capabilities for a Multikernel.* Master's thesis. ETH Zurich. http://www.barrelfish.org/publications/nevill-master-capabilities.pdf

[35] NIST National Vulnerability Database. 2014. CVE–2014–2523: netfilter: remote memory corruption. (March 2014). https://nvd.nist.gov/vuln/detail/CVE-2014-2523

[36] OpenMUL Foundation. 2015. OpenMUL Controller. (Sept. 2015). http://www.openmul.org/openmul-controller.html

[37] OpenStack Foundation. 2014. Neutron/SecurityGroups. (Aug. 2014). https://wiki.openstack.org/wiki/Neutron/SecurityGroups

[38] OpenStack Foundation. 2017. Identity API protection with role-based access control (RBAC). (Aug. 2017). https://docs.openstack.org/keystone/latest/admin/identity-service-api-protection.html

[39] OpenStack Foundation. 2017. Neutron/ML2. (April 2017). https://wiki.openstack.org/wiki/Neutron/ML2

[40] OpenStack Foundation. 2017. OpenStack Networking Guide. (Aug. 2017). http://docs.openstack.org/newton/networking-guide/

[41] Phillip Porras, Steven Cheung, Martin Fong, Keith Skinner, and Vinod Yegneswaran. 2015. Securing the Software-Defined Network Control Layer. In *Proc. NDSS.* https://doi.org/10.14722/ndss.2015.23222

[42] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. 2012. A Security Enforcement Kernel for OpenFlow Networks. In *Proc. HotSDN.* 121–126. https://doi.org/10.1145/2342441.2342466

[43] Robert Ricci, Eric Eide, and the CloudLab Team. 2014. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *;login:* 39, 6 (Dec. 2014), 36–38. https://www.usenix.org/publications/login/dec14/ricci

[44] Alexander Shalimov, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. 2013. Advanced Study of SDN/OpenFlow Controllers. In *Proc. 9th Central and Eastern European Softw. Engr. Conf. in Russia (CEE-SECR).* https://doi.org/10.1145/2556610.2556621

[45] Jonathan S. Shapiro, Eric Northup, M. Scott Doerrie, Swaroop Sridhar, Neal H. Walfield, and Marcus Brinkmann. 2007. Coyotos Microkernel Specification. (2007).

[46] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A Fast Capability System. In *Proc. SOSP.* 170–185. https://doi.org/10.1145/319151.319163

[47] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. 2009. *FlowVisor: A Network Virtualization Layer.* Technical Report OPENFLOW–TR–2009–1. OpenFlow Team. http://archive.openflow.org/downloads/technicalreports/openflow-tr-2009-1-flowvisor.pdf

[48] Seungwon Shin, Phillip Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. 2013. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Proc. NDSS.* https://www.internetsociety.org/sites/default/files/07_2_0.pdf

[49] Craig A. Shue, Andrew J. Kalafut, Mark Allman, and Curtis R. Taylor. 2012. On Building Inexpensive Network Capabilities. *SIGCOMM Comput. Commun. Rev.* 42, 2 (April 2012), 72–79. https://doi.org/10.1145/2185376.2185386

[50] Tom Van Cutsem and Mark S. Miller. 2013. Trustworthy Proxies: Virtualizing Objects with Invariants. In *ECOOP 2013 – Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, Giuseppe Castagna (Ed.). Lecture Notes in Computer Science, Vol. 7920. Springer, Berlin, Heidelberg, 154–178. https://doi.org/10.1007/978-3-642-39038-8_7

[51] Verizon RISK Team. 2013. 2013 Data Breach Investigations Report. (2013). http://www.verizonenterprise.com/resources/reports/rp_data-breach-investigations-report-2013_en_xg.pdf

[52] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX. In *Proc. USENIX Security*. 29–45. https://www.usenix.org/legacy/event/sec10/tech/full_papers/Watson.pdf

[53] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. OSDI*. 255–270. https://www.usenix.org/legacy/event/osdi02/tech/white.html

[54] Yan Zhai, Lichao Yin, Jeffrey Chase, Thomas Ristenpart, and Michael Swift. 2016. CQSTR: Securing Cross-Tenant Applications with Cloud Containers. In *Proc. SoCC*. 223–236. https://doi.org/10.1145/2987550.2987558