Pileus: Protecting User Resources from Vulnerable Cloud Services

Yuqiong Sun, Giuseppe Petracca, Xinyang Ge[*] and Trent Jaeger
Department of Computer Science and Engineering
The Pennsylvania State University
yus138, gxp18, xxg13, tjaeger@cse.psu.edu

ABSTRACT

Cloud computing platforms are now constructed as distributed, modular systems of *cloud services*, which enable cloud users to manage their cloud resources. However, in current cloud platforms, cloud services fully trust each other, so a malicious user may exploit a vulnerability in a cloud service to obtain unauthorized access to another user's data. To date, over 150 vulnerabilities have been reported in cloud services in the OpenStack cloud. Research efforts in cloud security have focused primarily on attacks originating from user VMs or compromised operating systems rather than threats caused by the compromise of distributed cloud services, leaving cloud users open to attacks from these vulnerable cloud services. In this paper, we propose the *Pileus* cloud service architecture, which isolates each user's cloud operations to prevent vulnerabilities in cloud services from enabling malicious users to gain unauthorized access. Pileus deploys stateless cloud services "on demand" to service each user's cloud operations, limiting cloud services to the permissions of individual users. Pileus leverages the *decentralized information flow control* (DIFC) model for permission management, but the Pileus design addresses special challenges in the cloud environment to: (1) restrict how cloud services may be allowed to make security decisions; (2) select trustworthy nodes for access enforcement in a dynamic, distributed environment; and (3) limit the set of nodes a user must trust to service each operation. We have ported the OpenStack cloud platform to Pileus, finding that we can systematically prevent compromised cloud services from attacking other users' cloud operations with less than 3% additional latency for the operation. Application of the Pileus architecture to OpenStack shows that confined cloud services can service users' cloud operations effectively for a modest overhead.

1. INTRODUCTION

Cloud computing has revolutionized the way we consume computing resources. Instead of maintaining a locally-administered data center, cloud users obtain resources on demand from a public cloud platform [3, 32]. Cloud vendors often construct their cloud platforms as a set of *cloud services* that implement users' operations. For example, in the OpenStack cloud platform, cloud services authenticate users, provision VMs, manage storage, etc. Often multiple cloud services collaborate to process a user's operation, forming a distributed, cloud computing environment.

One significant problem with this distributed computing environment is that cloud services themselves are complex software components prone to vulnerabilities. In the OpenStack, over 150 vulnerabilities have been reported in its cloud services, ranging from resource misuse [12] to authorization bypass [10] to the complete compromise of cloud nodes [1]. Further, current cloud platforms assume a flawed design where distributed cloud services fully trust each other [43]. Consequently, security breach in a single cloud service may allow adversaries to propagate attacks to other cloud services, producing security risks for any user's cloud resources.

Current defenses against cloud service vulnerabilities are often limited. First, OpenStack provides defenses to protect communications among services [28] and mechanisms for reducing user token privilege [2], but neither defense prevents a compromised cloud service from misbehaving or propagating attacks. Second, researchers have explored defenses to protect data security in clouds, including data encryption [31], data sealing [34] and protection against adversarial hypervisors and privileged domains [53, 9, 6, 46]. However, these systems aim to block cloud components from unauthorized access to user data, but cloud services often need access to user data to perform operations. Third, researchers have explored defenses to mitigate compromises of certain cloud components [45]. These systems often only address compromised compute services and require all other cloud services to be trustworthy. Finally, researchers have explored approaches to better protect distributed web applications deployed on PaaS clouds [29, 5]. These systems focus on protecting data security for cloud-hosted applications, relying on the cloud platform, including the underlying cloud services to be trustworthy.

The goal of this work is to prevent malicious users from gaining unauthorized access to other cloud users' resources by exploiting vulnerabilities in cloud services. To achieve this goal, we leverage the following insights. First, we find that many cloud services in the OpenStack cloud platform run in a *stateless* manner, where cloud services do not maintain internal state across user operations. Thus, we propose converting cloud services into a set of stateless *event handlers* [15] that are spawned on demand to process individual user's operations with only that user's permissions. Second, by applying *decentralized information flow control* (DIFC) [24] model, we localize the security decisions made by cloud services to a few trusted services, thereby preventing adversaries from exploiting vulnerabilities that may be present in those cloud services to access users' data. While the DIFC model has been applied to distributed systems [52] and even to control user VMs on the

[*] Now at Microsoft Research.

ACSAC '16, December 05-09, 2016, Los Angeles, CA, USA
© 2016 ACM. ISBN 978-1-4503-4771-6/16/12. . . $15.00
DOI: http://dx.doi.org/10.1145/2991079.2991109

cloud [29, 5], a cloud platform based on spawning event handlers on demand to service user operations presents special challenges to: (1) restrict how cloud services may be allowed to make security decisions; (2) select trustworthy nodes for enforcing access controls properly in a dynamic, distributed environment; and (3) limit the set of nodes a user must trust to service each operation.

We address these challenges by designing the Pileus cloud service architecture, which governs the execution of users' cloud operations across a distributed cloud platform. When a user performs a cloud operation, a *Pileus initiator* configures access control for the operation's execution in the cloud, only allowing approved programs when invoked in an operation-specific manner to exercise the user's authority (e.g., declassify or endorse user data). To enforce the user's access control across cloud nodes throughout the operation, the *Pileus ownership registry* selects the cloud nodes deemed most capable of enforcing the user's access control throughout operation execution. On each node, a *Pileus daemon* configures access control enforcement to manage the spawned event handlers. We demonstrate Pileus by porting OpenStack to the Pileus cloud architecture. We show how the OpenStack cloud services naturally comply to the Pileus cloud architecture, and how we implement OpenStack cloud operations in Pileus OpenStack. Results show that we can improve the security of OpenStack in a systematic way by factoring existing OpenStack services into event handlers, resulting in no more than 3% additional latency on operation execution as perceived by cloud users.

We highlight the following contributions of this paper:

- We define the *Pileus cloud architecture* for preventing malicious users from exploiting vulnerabilities in cloud services to obtain unauthorized access to other users' resources by spawning a set of stateless event handlers to process each user's operation with only that user's permissions.
- Pileus confines event handlers where: (1) a *Pileus initiator* configures access control at operation initiation; (2) a *Pileus ownership registry* selects cloud nodes to run event handlers that are deemed most capable of enforcing user's access control; and (3) *Pileus daemons* on each cloud node spawn event handlers and govern their execution.
- We have ported OpenStack cloud services to Pileus. We show how Pileus OpenStack systematically prevents the exploitation of cloud service vulnerabilities in OpenStack to protect cloud user data for low overhead.

2. PROBLEM DEFINITION

2.1 Security Problems in Cloud Services

Consider a simplified cloud with only the cloud services running as shown in Figure 1. In this example, user Alice and user Bob share the same cloud platform (i.e., same set of cloud services), but are mutually distrustful. Consider the following cloud operations performed by Alice and Bob.

1. Alice takes a snapshot of her running VM, then terminates the VM. The used volume of the VM is returned to the volume store. The snapshot is saved to the image store.
2. Bob requires a new volume from the volume store and attaches it to his running VM.
3. Alice requires an image from image store and launches a new VM based on it.

Below, we identify some security problems that may occur when performing these cloud operations on current cloud platforms, using the OpenStack cloud platform [27] as the example.

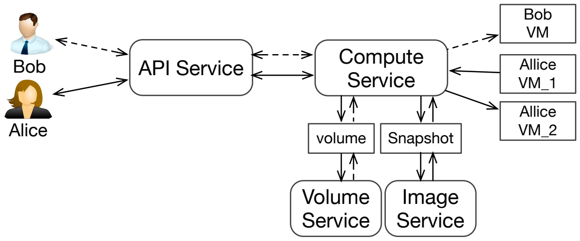


Figure 1: A simplified cloud platform. Rounded rectangles represent cloud services, rectangles represent cloud resources. Solid arrows represent information flows as a result of Alice's cloud operations and dashed arrows represent information flows as a result of Bob's cloud operations.

A basic data security policy is *cloud user isolation*. For example, VMs owned by Alice and Bob should be kept secret from each other. However, users may exploit vulnerabilities in cloud services to bypass even this basic policy. As an example, a vulnerability in the OpenStack API service [10] allows Alice to bypass the authorization and take a snapshot of Bob's VM in operation #1 above. Several similar vulnerabilities [39, 40] were found in OpenStack cloud services, leading to unauthorized access to cloud resources of various kinds.

Cloud services also fail to enforce *data secrecy* over users' resources. For example, when Alice returns an used volume, the volume service should ensure that sensitive data on the volume is removed. However, vulnerabilities [12, 13] were found in cloud services that may cause the volume erasure to be omitted or bypassed. Thus, when Bob obtains a volume used by Alice previously, as in operation #2, he may learn some of Alice's sensitive data.

In addition, cloud services may fail to enforce *data integrity* over users' critical cloud resources. For example, cloud platforms enable their users to specify an image list, listing images that users have approved for use (i.e., approved the integrity of). Such integrity protection aims to prevent users from accidentally using corrupted images provided by adversaries. Unfortunately, vulnerabilities were found in cloud services that would subvert this protection. For example, a vulnerability [41] in the image service allows Bob to manipulate Alice's image list to insert his maliciously-crafted VMs. As a result, when Alice performs operation #3, she may be tricked into launching a VM using Bob's malicious image.

Other security issues may stem from the insecure design of the cloud platform at large. Current cloud platforms assume a trusted computing base that includes all cloud services. Thus, if a single cloud service becomes compromised such that an adversary can control the messages produced by the service, it can easily propagate attacks to other services. For example, multiple attacks [43, 44, 45] were shown that enable adversaries to cause a cloud service to *forge requests* to other cloud services to perform operations specified by the adversaries.

2.2 Problem Definition

From the above analysis, we identify three main problems of current cloud platforms: First, cloud services run with too many permissions. Current cloud services can act on behalf of any and all cloud users. Consequently, vulnerabilities in these cloud services enable confused deputy attacks [17]. Second, cloud services make security decisions. Current cloud services perform a variety of access control checks and even define the access control policies to be enforced in some cases. Such checks can often be bypassed or omitted and policies may not reflect user requirements, leading to unauthorized data access. Third, cloud services fully trust each other. An adversary can forge requests to other cloud services via cloud service messaging and trick them into performing arbitrary operations. From a user's perspective, every cloud service must be trusted to protect the user's security, even if the user is not using that cloud service.

Previous cloud defenses do not address these problems. Systems such as CloudVisor [53], Self-Service Clouds [9] and Haven [6] focus on protecting user data against adversarial hypervisors and privileged domains. Systems such as Cloud Verifier [35] and Excalibur [34] ensure that only cloud nodes that satisfy certain properties may access user's data. In both cases, such defenses prevent untrusted code from accessing security-critical data, but cloud services must process user requests while protecting each user's data. Thus, these defenses do not prevent cloud services from being leveraged as confused deputies nor do they prevent adversary-controlled cloud services from propagating attacks.

These problems represent some of the core challenges in building secure systems in general. For example, Asbestos [15] studied how multi-user programs (e.g., web servers) can be confined to run with least privilege. DStar [52], Fabric [23] and Mobile-fabric [4] studied how distributed computation can be carried out in a system where its components do not trust each other. The decentralized information flow control (DIFC) model [24] and systems [51, 21, 15, 33, 11], as a general approach, show how security of complex applications can be factored into small and simple programs.

However, the problem for cloud services is that a set of cloud services need to work together to implement individual operations on demand isolated from the rest of the system. In a sense, we need to construct *assured pipelines* [7] with operation-specific restrictions on each pipeline. In addition, these assured pipelines may leverage arbitrary cloud nodes chosen dynamically, and we need to ensure that the cloud nodes chosen are capable of enforcing the access control requirements. DIFC systems do not enforce either operation-specific constraints nor evaluate the trustworthiness of nodes based on their workloads.

3. DIFC MODEL BACKGROUND

Pileus adopts its definitions of *security labels* and *ownerships* from the DIFC model in Flume [21] and its definition of *message labels* from DStar [52]. Readers who are familiar with DIFC may skip this section.

Security Labels. DIFC models define security labels in terms of sets of tags. Tags are random identifiers with no inherent meaning until they are assigned to labels. Each process runs with two labels, S for secrecy and I for integrity. If tag $a_s \in S$ for a process (e.g., cloud service), the process is assumed to hold secrets only accessible to processes with security labels containing the tag a_s. Similarly, if tag $a_i \in I$, then that process is endorsed by the creator of tag a_i. Labels form a lattice under the partial order of the subset relation among tags [14]. Data objects in the cloud (e.g., images, VMs) are also assigned security labels.

Consider an information flow from a source p to a destination q[1]. The DIFC constraint that protects the secrecy and integrity of any information flow $p \rightarrow q$ is:

$$S_p \subseteq S_q \text{ and } I_q \subseteq I_p \tag{3.1}$$

Ownerships. DIFC models may express trust in processes to make some security decisions on behalf of tags, calling the set of tags in which a process is trusted *ownerships*. Functionally, an ownership allows a process holding the ownership to adjust its security label by adding and removing such tags. For example, ownership of secrecy tag a allows a process to remove a from its secrecy label, trusting the process to declassify its data associated with tag a. Ownership of integrity tag b allows a process to add b to its integrity label, effectively trusting the process to endorse data (e.g., input messages) to satisfy the integrity requirements associated with tag b. Note that ownerships may be transferred between processes, allowing one process to delegate authority to another.

For secrecy, p would get the maximum latitude in sending data to q if it lowers its secrecy label to $S_p - O_p$ and q raises its secrecy label to $S_q \cup O_q$. In this case, the DIFC constraints for safe information flows are:

$$S_p - O_p \subseteq S_q \cup O_q \quad and \quad I_q - O_q \subseteq I_p \cup O_p \tag{3.2}$$

Message Labels. When p and q run on different hosts (e.g., cloud nodes), no single reference monitor can see the labels of p and q at the same time. In this case, information flow is transitively enforced using *messages labels*. Say m is a message sent from p to q. By attaching a message label to m, the information flow constraint 3.2 thus becomes the following:

$$S_p - O_q \subseteq S_m \subseteq S_q \cup O_q \tag{3.3}$$

The left half of above constraint $S_p - O_p \subseteq S_m$ is enforced on p's node and the right half $S_m \subseteq S_q \cup O_q$ is enforced on q's node. The integrity constraint is similar. Thus two hosts can work collaboratively to enforce the information flow constraint.

Example. Figure 2 illustrates some uses of the DIFC model in addressing problems introduced in Section 2.1. Figure 2(a) shows that by confining cloud services to their respective users' secrecy labels, Alice ($S = \{a\}$) and Bob ($S = \{b\}$) can only access their own VMs. Such isolation would be *end-to-end*, restricting cloud service interactions as well (e.g., Alice's services cannot forge messages for Bob's services or modify Bob's cloud resources). Figure 2(b) shows that when Alice wants to release a used volume to public, the volume must first be sanitized by a declassifier whose trust is conferred by running with Alice's ownership. Similarly, Figure 2(c) shows when Alice wants to use a public image provided by Bob, the image must first be verified by an endorser whose trust is conferred by running with Alice's ownership.

4. PILEUS DESIGN

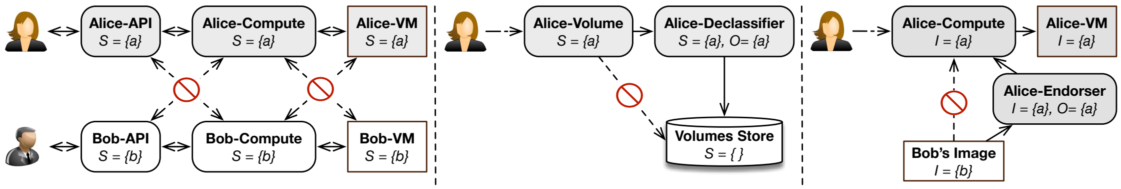
4.1 Pileus Overview

Based on the problems highlighted in Section 2.2, we identify the following security goals for the Pileus design.

- **Minimize Permissions**: Restrict event handler permissions to only those necessary for executing the handler for the specific operation, typically the permissions of the user requesting the operation.
- **Minimize Security Decisions**: Restrict security decisions to only those cloud services that are trusted by the users to manage their data security.
- **Eliminate Dependence on Untrusted Nodes**: A users' data security should never depend on cloud nodes that the user does not trust. Further, when performing an operation, a user should delegate his trust to the node that has the least likelihood of compromise.

To achieve these goals, we leverage the following insights. First, we can spawn cloud services on demand as *event handlers* that run with only the permissions necessary for that command. The event handler abstraction was proposed for the Asbestos system [15],

[1] p and q can be processes or data objects, but they cannot be data objects at the same time.



(a) Alice's cloud services and data are _isolated_ from Bob's

(b) Cloud services cannot release Alice's data to public unless _properly declassified_

(c) Cloud services cannot read public data unless _properly endorsed_

Figure 2: Decentralized information flow control examples in Pileus.

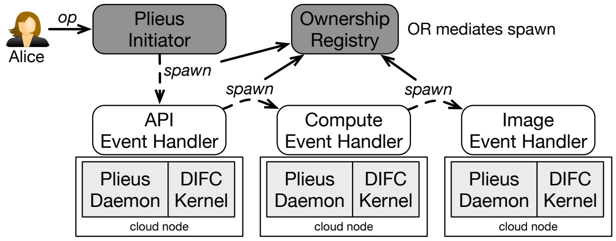


Figure 3: Overview: a user operation performed on Pileus.

where stateless services are launched with minimal permissions. The Pileus design leverages this insight because we find that cloud services are already architectured as stateless services, although we need a platform to spawn such event handlers dynamically in a manner that complies with the user's security policy. Second, researchers have proposed the *decentralized security principle* [4], which states that the security of a principal must not depend on any part of the system that it does not trust. The Pileus design targets this goal by choosing nodes most capable of enforcing the user's security policy to run event handlers, and prevent nodes not trusted by a user from tampering with the user's data security. We leverage prior work in secure capability systems to prevent depending on nodes that would violate security constraints [16] (i.e., could be an adversary). Third, the *decentralized information flow control* [24] (DIFC) enables access control that expresses a subject's (e.g., event handler) authority over security decisions explicitly as ownerships (see Section 3). The Pileus design leverages this insight to restrict the security decisions to only event handlers that are trusted by the user to manage their data security.

Pileus utilizes these insights to implement user operations as follows, shown in Figure 3. First, when a user submits an operation to the cloud, the *Pileus initiator* validates the user's identity and computes the access control requirements for executing the operation, as described in Section 4.3. Second, each user operation is executed as a set of event handlers launched on-demand, corresponding to the cloud services needed to complete the operation on the old cloud platform. Starting with the Pileus initiator, requests to select nodes on which to spawn new event handlers are submitted to the *Pileus ownership registry*, which selects a node that satisfies the cloud security policy and minimizes the risk of attack from other users, as described in Section 4.4. Third, each node is empowered with the ability to enforce DIFC policies securely to govern the execution of all event handlers. When a node receives a request to spawn an event handler, it validates that it has been approved to run the specific event handler by obtaining an *authority token* granting such as capability from the OR. When a node completes the execution of its event handler, it no longer needs the authority to access user data, as described in Section 4.5.

The remainder of this section details the key design tasks for building a Pileus system. The foundation of Pileus security is the spawn protocol, described in Section 4.2, which determines how to distribute the authority to enforce DIFC policies over every event handler while preventing unauthorized access to user data. The other design tasks involve best effort methods to restrict the event handlers that may make security decisions (Section 4.3), to select nodes that satisfy a cloud security policy while minimizing risk (Section 4.4), and to enable individual cloud nodes to revoke authority from their delegatees (Section 4.5).

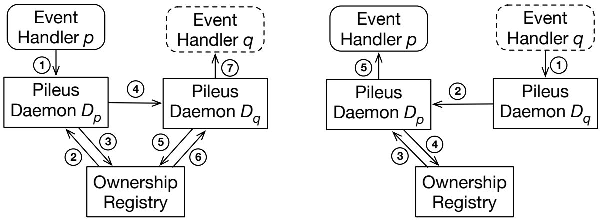
Security Model. In Pileus, we assume the trustworthiness of two global services: a *Pileus initiator* that authenticates users and computes access control requirements for their operations, and a *Pileus ownership registry* that manages the authority distribution. Since these services are relatively static and simple, we expect them to be fully trusted by any user in cloud. We trust the cloud vendor at the organizational level.

The local enforcement mechanism on each cloud node consists of a *Pileus daemon* and a *DIFC kernel*. A user trusts those services on any cloud nodes that run her cloud services. However, we do not assume the mutual trust between enforcement mechanisms on different cloud nodes; that is, we assume individual cloud nodes may be under the complete control of an adversary. Thus, the TCB of a cloud user will include the globally trusted services and the local enforcement mechanisms to which *the user has delegated her authority*. The attacks we aim to block are exemplified in Section 2.1. We assume an adversary may launch confused deputy attacks or gain complete control over cloud services by exploiting vulnerabilities in them. He may further escalate his privilege on a cloud node (e.g., by exploiting a kernel vulnerabilities). Pileus's approach for mitigating adversarial cloud services and nodes is to enforce the decentralized security principle [4]: a user's data security does not depend on any components of the cloud platform that are not part of her TCB.

4.2 Pileus Spawn Protocol

The execution of a user operation in a Pileus cloud is implemented by spawning a sequence of event handlers for each program necessary to complete the operation. Thus, the security of operation execution is governed by the protocol to spawn event handlers to execute user operations, the *spawn protocol* shown in Figure 4(a). In the spawn protocol, a cloud node wishing to spawn an event handler on another cloud node, the *parent node*, presents evidence of its authority over a user operation (called an *authority token* below) and the program to be spawned to the *Pileus ownership registry* (OR). The OR completes the spawn protocol by choosing a *target node* to execute the specified event handler and producing evidence that the target node can also act on behalf of the user operation (i.e., access user data on that node and make further spawn requests on behalf of the user).

The aim of the spawn protocol is to prevent: (1) nodes that lack a user's authority from spawning event handlers that may access that user's data and (2) nodes that fail to satisfy a cloud security policy (regarding the user's trust in the node to enforce her access control) from being selected (by the OR) as target nodes or given



(a) Spawning an event handler (b) Ownership revocation

Figure 4: Protocols of ownership delegation and revocation.

the authority to execute the user's handler[2].

In Pileus, authority over a user's operation is distributed to cloud nodes as *authority tokens*, $t = \{own, n, auth, sig\}$, where: (1) own are the DIFC ownerships describing the authority of the token, usually the user's DIFC label; (2) n is the node for which the token applies; (3) $auth$ describes the conditions under which an event handler may obtain ownerships for this operation, which we call *ownership authorizations* (see Section 4.3), to control how ownerships may be used in execution of the operation; and (4) sig is an OR signature over the first three elements that certifies the token. In effect, when an OR grants an authority token to a target node, this implies that the user trusts the node to protect her data security requirements (i.e., enforce her DIFC policy) for the execution of this operation. Each cloud node runs a *Pileus daemon* which obtains, propagates, and revokes (see Section 4.5) authority tokens for each node.

In Pileus, the OR produces authority tokens for a target node only when the distribution of a user's trust to that node complies with a cloud security policy. Such cloud security policies could prevent mutually distrustful users from running handlers on the same node (i.e., conflict-of-interest policies), could prevent handlers from being deployed on nodes that ran privileged handlers on behalf of other users (i.e., information flow policies), and could prevent the use of nodes that have run too many handlers or handlers on behalf of too many users (i.e., cardinality policies). Traditionally, capability systems (like DIFC) allow the holders of authority to choose how they propagate authority, but this may present problems in the cloud because a compromised or ignorant cloud node may violate the cloud security policy that protects the user's operation. This problem is analogous to the classic capability system problem caused because the authority to exercise a capability also permits the authority to delegate capabilities, leading to violations of the $\star$-property [49]. Thus, we leverage the solutions proposed to solve the classical capability system problem [19, 38, 16], in particular *identity-based capabilities* in the ICAP system [16] that associate authority with individual nodes and validate the propagation of such authority.

Figure 4(a) shows how an event handler dynamically spawns another event handler. When a handler p needs to spawn another event handler, its request (i.e., event) will be proxied by its Pileus daemon D_p (Step 1). D_p will send the spawn request to the OR (Step 2). In its spawn request, D_p must specify the authority token t that it would like to delegate. This reflects the fact that spawning event handlers on a new node requires a user's trust in that new node. After receiving the spawn request, the OR checks in its *ownership graph*, a data structure that tracks the states of cloud nodes, to see if D_p actually holds the user's ownership. If so, the OR will select a target node by running the spawn scheduling algorithm as discussed in Section 4.4 and update its ownership graph to mark the ownership delegation. It will also produce a new authority token for the target node that incorporates its identity. The OR then returns a reference to the new authority token as well as the target node identity to D_p (Step 3). Assume the Pileus daemon running on the target is called D_q. D_p will then send p's event, q's security label (inherited from p) and the authority token reference to D_q (Step 4). D_q will query the OR to retrieve the authority token created for it (Step 5 and 6). It then validates if the authority token is sufficient to create q's label. If so, D_q launches q with specified label and may or may not delegate ownerships to q depending on whether or not q meets the ownership authorizations specified in the authority token (Step 7), as discussed in Section 4.3. As q is run, the Pileus kernel and other Pileus enforcement mechanisms on q's node ensure that all q's data accesses comply with the DIFC policy.

It is easy to prove that this spawn protocol satisfies the expected security properties. First, it prevents a parent node that lacks a user's authority (per the OR) from spawning event handlers that may access that user's data. If a parent node provides an authority token that does not correspond to the authority tokens stored by the OR, then the delegation request will be rejected. Further, since nodes can only access user data on their node or by spawning an event handler on another node, the parent node will not be able to trick the target node into access user data on its behalf. This ensures the decentralized security principle. Second, no node that fails to satisfy a cloud security policy will obtain an authority token for that user's operation. The OR checks the cloud security policy when selecting a target node and the OR validates the cloud security policy when producing an authority token. This ensures that cloud security policy will always be met. In addition, the OR uses a best effort protocol to select targets to lower the risk of choosing a compromised node, as described in Section 4.4.

As an optimization, the parent node needs not request the identity of a target node from the OR each time. A parent node may cache a set of approved nodes or reuse nodes from previous spawn requests for that operation. In this case, the parent node may submit an event and its authority token (in lieu of a reference to a new token) in a spawn request to a target node optimistically to reduce messaging. The target node will still only be allowed with the user's authority should an OR provide a authority token for that user to the target node. In this case, the spawn request from the parent node must be signed to enable the OR to verify the source and include a nonce for replay protection.

4.3 Restricting Security Decisions

In Pileus, security decisions, such as declassification and endorsement of cloud objects, are localized to certain event handlers that a cloud user trusts to manage their data security. For example, Figure 2(b) shows that a volume declassifier is trusted by Alice (e.g., Alice-Declassifier) to remove secrets before releasing the volume to public, and to declassify the volume this declassifier runs with Alice's ownership. Since event handlers are dynamically spawned, one way for a trusted event handler to obtain ownership is by inheriting the ownership from its parent (e.g., Alice-Volume). However, this defeats the purpose of DIFC, as we want to localize the authority to make security decisions to only the event handlers the user trusts. Other event handler are confined so that they cannot violate the user's data security either inadvertently or intentionally. As a result, we need a mechanism in Pileus that authorizes the invocation of event handlers with user ownerships.

This problem is analogous to the *setuid* problem of UNIX systems, where certain programs (e.g., /usr/bin/passwd) are allowed to escalate privileges on invocation (e.g., run as the `root` user), whereas others are confined to user permissions. Current methods

[2] The OR may also prevent a node from being selected as a target node for functional reasons.

to allow escalation of privileges on invocation are either inflexible or fail to prevent a compromised parent process from exploiting ownerships. One approach is to statically associate ownerships to event handlers, as adopted by some DIFC systems [21, 11]. However, we do not want declassifiers to be able to run with ownerships of *all the users* that have approved its use. Alternatively, systems use the security labels of the parent process and/or the program to be executed to determine its authority, such as SELinux [26] transition rules. However, simply checking the label of a program file does not prevent a compromised event handler from running modified binaries, applying ownerships to the wrong objects, or invoking programs with ownership for the wrong operations.

Pileus includes a mechanism for computing legal delegations of a user's ownership to an event handler by relating the invocation of an event handler approved to hold such ownership to a specific operation from that user. When an operation is received, Pileus maps the operation to a set of authorized delegations that are allowed for the operations execution. These authorized delegations are called *ownership authorizations*, which are tuples of the form $auth = (h(e), own(e), args(e))$, where: (1) $h(e)$ is the hash of the event handler e's code; (2) $own(e)$ are the ownerships to be delegated to the event handler e; and (3) $args(e) = \{(arg_1, ind_1)...(arg_n, ind_n)\}$ is a specification of specific argument values arg_i and index values ind_i input to the event handler e. Only if the program corresponding to the hash value $h(e)$ is invoked by the user whose tags correspond to $own(e)$ with the arguments specified by $arg(e)$ does the resulting event handler obtain that user's ownerships.

Pileus constructs ownership authorizations primarily from user operations. When a user runs a operation, Pileus records the user's ownerships and maps the operation's arguments to the expected arguments for the specific invocations. Operation-specific rules are necessary to define these mappings, but fortunately, there are not a large number of event handler invocations that require ownership. The identity of approved endorsers and declassifiers for particular operations must be provided ahead of time by the users, but this decision may be aided by a trusted third party chosen by each user. To return a volume as shown in Figure 2(b), Alice only needs to specify the hash of the volume declassifier that she trusts, $h(vd)$. The secrecy label of the volume declassifier and the argument it is spawned with (in this case, the volume ID to release) will be inferred from Alice's operation submitted to the cloud, producing a ownership authorization $T_a = (h(vd), \{a\}, \{volume\text{-}id[INDEX]\})$.

4.4 Selecting Nodes for Spawning Handlers

In Pileus, spawning an event handler on a node means that the user trusts the chosen cloud node (i.e., its kernel and Pileus daemon) to protect her data security. This trust may be misplaced if the node is not capable of protecting the user's data security because it may include adversarial code in its trusted computing base or be compromised.

Since we do not assume the ability to detect adversarial nodes, in Pileus, we devise a *spawn scheduling algorithm* that allows cloud to act on behalf of users to select cloud nodes that are most likely to be capable of protecting the user's data security. This algorithm leverages users' security constraints (e.g., a conflict-of-interest policy among users) and functional constraints of cloud vendor to find legal nodes for spawning an event handler for a user. In addition, the spawn scheduling algorithm aims to select nodes in a manner that minimizes the additional likelihood that a user's operation may be compromised given the current state and history of the node.

```
Input: Ownership graph $G$, ownership to delegate $o$, and cloud policy $P$
Output: Selected cloud node $n$
for $i \in$ TCB($o$, $G$) do            ▷ Set of available node that already hold $o$
    if $i$ meets $P$ then               ▷ A cloud node that satisfies cloud policy
        $n \leftarrow i$
        $return$
    end if
end for
$\Phi \leftarrow$ SORT_BY_CURRENT($G$)       ▷ Sorted nodes by current users
for $S \in \Phi$ do
    $S' \leftarrow$ SORT_BY_HISTORY($S$, $G$)  ▷ Further sort nodes by history
    for $k \in S'$ do
        if $k$ meets $P$ then
            $n \leftarrow k$
            $return$
        end if
    end for
end for
```

Figure 5: Spawn scheduling algorithm

The foundation of the spawn scheduling algorithm is a data structure that represents the state of the cloud nodes running event handlers, which we call the ownership graph. An *ownership graph*, $G = (V, E)$, where: (1) vertices $v \in V$ are the set of ownerships granted per cloud node, $v = (o, n)$, where u is a user's ownership and n is a node and (2) edges $(u, v) \in E$ are the ownership delegations from vertex u on one node to vertex v on a second node. The ownership graph enables us to reason about two main things: (1) the set of nodes that are already part of a user's TCB by virtue of having been delegated with that user's ownership and therefore trusted to protect that user's data security and (2) the set of ownerships that must be managed by each node in order for that node to protect all those users' data security requirements from attacks from other nodes.

The spawn scheduling algorithm is illustrated in Figure 9. The algorithm takes the ownership graph G, the particular ownership to be delegated o, and the security and functional policies for the cloud P. First, Pileus will always try to spawn event handlers on cloud nodes that are already part of a user's TCB (i.e., the same cloud node or other nodes that currently hold the user's ownership). Only if these nodes are not available (e.g., they do not meet cloud policies), will the event handler be spawned on another cloud node because such as choice would expand a user's TCB.

If a new cloud node must be introduced into a user's TCB, this node must also satisfy the cloud policy P and must minimize the likelihood of compromise. Two metrics are considered to minimize the likelihood of compromise. First, Pileus will try to spawn on a node that least number of users are currently using. A best case scenario is that the user will be the only one on the node. Second, Pileus will try to spawn on a node with shortest history (i.e., least number of users have used the cloud node since its last reboot). This reduces the likelihood of the node being "polluted" (e.g., has backdoors implanted by attackers) by other users. Ideally, the cloud node is freshly installed from a clean state[3].

4.5 Revoking Authority

Spawn may expand a user's TCB to new cloud nodes. However, we do not want the cloud node to be in a user's TCB forever. Thus, it is important to revoke the user's ownership from a cloud node once the node is no longer involved in serving the user's operation, reducing the attack window available to adversary if the node becomes compromised.

One challenge is to figure out when a cloud node is no longer involved in executing a user's operation. Fortunately, it is straightforward when using event handlers in the cloud. Since cloud ser-

[3] In Pileus, we periodically re-image a cloud node to restore it to a pristine state, using a mechanism similar to cloud verifier [35].

vices are stateless, one request will have only one response. Thus, when an event handler returns a response, it indicates that the event handler has completed its job. It is then safe to revoke the user's ownership for running that event handler from that cloud node. In case an event handler never returns (e.g., due to node failure, or an adversary trying to extend its attack window), a request timeout can be used as a signal of revocation of [4]. However, if a node causes timeouts frequently, the cloud vendors may detect this behavior as an anomaly. Using event handler completions for revocation enables Pileus to reduce the temporal attack window for adversaries to compromise a node running a particular user's service.

The revocation protocol is shown in Figure 4(b). The revocation protocol in Pileus leverages the response generated by an event handler q as a signal of revocation. After q completes, it would return a single response to p (step 1), and response is proxied by D_q to D_p (step 2). After D_p receives the response, it will send a revocation request to the OR (step 3), revoking any authority delegated to the target node via D_q. The OR validates the revocation request using its ownership graph, and it removes the edge along with the ownerships held by the target node in the graph. In addition, the OR will also purge any authority tokens that are associated with the delegation. This effectively invalidates the ownership held by the Pileus daemon D_q, so D_q will no longer be able to spawn event handlers on other cloud nodes using that authority token. The OR confirms the revocation (step 4), and D_p proxies the response back to the requesting event handler p (step 5). Compared with an expiration based strategy [52], revocation in Pileus is more *timely*. Authorities are revoked from a cloud node immediately after the event handler running on top completes its job, leaving no unnecessary cloud nodes within a user's TCB.

5. IMPLEMENTATION

The Pileus implementation consists of introducing the new, trusted Pileus services (Section 5.1), decomposing OpenStack cloud services to run on Pileus (Sections 5.2 and 5.3), and ensuring Pileus can enforce isolation among users and between handlers and each node's privileged processes (Section 5.4).

5.1 Trusted Services

In Pileus, there are two globally trusted services: the Pileus initiator and the Pileus ownership registry. The Pileus initiator serves as a trusted portal for cloud users to initiate their cloud operations. It (1) collects the declassifier or endorser that the user wants to use in her operation and maps it to the corresponding hash; (2) infers the input arguments to the declassifier or endorser from the user's operation and (3) maps the user's credentials to corresponding tags. It then constructs the ownership authority for the user's operation and collaborate with the ownership registry to spawn the first event handler (often event handler of the API service) for the user's operation. Its code base is ∼500 SLOC. The Pileus ownership registry maintains the ownership graph, and runs the spawn scheduling protocol to ensure that the delegation of user's authority meets cloud policy. Its code base is ∼1200 SLOC. Both Pileus initiator and ownership registry are implemented in Python.

5.2 Event Handlers

The existing code for the OpenStack cloud services forms the basis for the event handlers in Pileus-OpenStack. Our observation is that OpenStack cloud services[5] are constructed using an event-dispatch loop. When an event arrives at a cloud service, it is dispatched to a worker thread for processing that event. Across events, the cloud services retain no persistent state, freeing each worker thread to run independently on one event, just as we require for event handlers. We extract the worker thread code from the cloud services that leverage this event-dispatch loop design to create event handlers. Each event handler is invoked as part of a single cloud operation performed by a single user, enabling Pileus to confine them to the user's security label.

Event handlers may run helper programs, such as legacy Linux utilities (e.g., qemu-img), to complete their processing on the host. These helper programs are ephemeral as well—they are executed to complete one job and exit after the job's completion. In Pileus, helper programs' processes inherit their labels from the event handlers that invoked them, so they are confined by the Pileus.

The advantage of DIFC approach is that we did not need to understand all of OpenStack's code. Majority of event handlers are decomposed as-is from worker threads of cloud services. Only a few required modification in order to invoke declassifiers or endorsers or to interact with the database, as we will discuss later. The total modifications we made to event handlers constitute ∼300 SLOC out of the ∼120,000 SLOC (not counting Python libraries used) for the 10 OpenStack cloud services in our deployment.

5.3 Pileus Daemon

Events in OpenStack are implemented in either one of two forms: HTTP requests or messages from the message queue. Thus, cloud services often share the same underlying implementation of the dispatcher loop—different types of cloud services only differ in the specific implementation of event handlers. Thus, the bulk of Pileus Daemon implementation comes directly from the dispatcher loop of cloud services. We mainly augmented it with the ability to manage labels and ownerships for event handlers and to execute the spawn and revocation protocols discussed in the design. The total code base for Pileus Daemon is ∼5000 SLOC, out of which ∼3,800 SLOC originates from the dispatcher loop of OpenStack cloud services and ∼1,200 SLOC are our additions.

5.4 Pileus Enforcement Mechanism

Pileus Kernel. Most of the cloud objects that are managed by cloud services are *files* (e.g., images, VM disks). For Pileus, we built a DIFC kernel to enforce access control over these objects on each cloud node. The DIFC kernel is mainly implemented as a kernel security module, leveraging the Linux Security Module hooks [50]. Unlike Flume [21], we did not modify the system call interface. Instead, we implemented the label and ownership operations through an interface similar to device drivers. A user space library is created for exposing an API abstraction to Pileus Daemon and DIFC-aware event handlers. The kernel module is ∼9,000 SLOC and the userspace library is ∼1,200 SLOC.

Network Namespace. Network Service (e.g., nova-network) configures *network objects* such as Linux bridges, software switches, and iptables, on a cloud node, in order to manage networking for VMs. These network objects are challenging for access control since they are kernel resources which are traditionally treated as single objects by the kernel from an access control perspective. Thus, a vulnerability in an event handler may enable one user to modify the network configuration of another user. For example, inappropriate processing of firewall rules [42] allowed one user to

[4] Current cloud platforms use request timeouts to detect node failures.

[5] In this work, we focused on 10 cloud services as a proof-of-concept: nova-api, nova-scheduler, nova-conductor, nova-compute, nova-network, glance-api, glance-registry, cinder-api, cinder-scheduler and cinder-volume. These services implemented the core functionality of a cloud platform. The rest OpenStack cloud services are designed in a similar fashion that will plan to include in future.

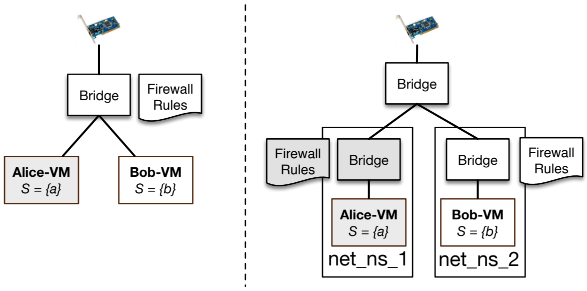


Figure 6: Isolating networking objects via network namespace.

block the network connection of another. To mitigate such vulnerabilities, Pileus must isolate each user's network objects in order to provide effective access control.

To achieve this goal, we leverage the *network namespace* abstraction of the Linux kernel in our implementation. A network namespace is logically a separate copy of the network stack, with its own network devices, routes, and firewall rules. Network objects can be isolated into different network namespaces. By running each event handler with access only to its user's network namespaces, the Pileus kernel restricts each event handler to its user's networking objects. Figure 6 shows an overview of our implementation for nova-network. Each VM is attached to a private Linux bridge that runs in its own network namespace. To access physical network, the two private bridges are connected to a host bridge via `veth` pairs. The host bridge runs in the native network namespace and contains physical network interface as one of its ports. When an event handler specifies a firewall rule, the rule will be applied to the private bridge instead of the host one, therefore eliminating its potential effects on other user's VMs.

Database. OpenStack relies on legacy database servers to store *database objects* (e.g., metadata of user such as their SSH keys). To extend information flow control to the database objects, we rely on the security framework built inside of the database servers. SEPostgreSQL [36] assigns labels to database objects and enforces access control over all requests to database objects. For each access request, the label of the requestor as well as the database objects are passed to a security server (the Pileus kernel in our case), which performs the access decision, enabling consistent enforcement of access control for system and database objects. The downside of this approach is, however, the database server must be fully trusted, since it runs the enforcement mechanism.

In order to obtain the label of the requestor (i.e., the querying event handler), we disable all remote connections to the database. Instead, an event handler requesting database access must spawn a special event handler, called *DB-Client*, on the cloud node that hosts the database server. The DB-Client will inherit its parent's label through the Pileus spawn mechanism, and connect to the database server via a Unix domain socket. The database will obtain the DB-Client's label from the socket descriptor (e.g., using the *getpeercon* API in libselinux [22]). This prevents an adversarial cloud node from accessing arbitrary data from the database server, since an adversarial node can only spawn DB-Clients with the labels of users for which it holds ownerships.

Libvirtd. In OpenStack, *VM objects* (e.g., VMs, containers) are managed by virtualization drivers which are often daemon programs that run with `root` privilege. One example is libvirtd. The question is how to extend access control to these VM objects, preventing these daemons from being leveraged as confused deputies (e.g., In current OpenStack, Bob's event handler may ask libvirtd to operate over Alice's VM).

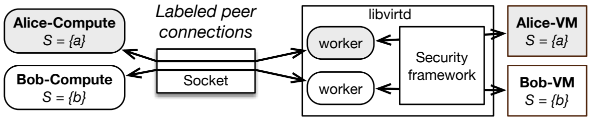


Figure 7: Information flow between event handlers of compute service and libvirtd daemon.

	CVE ID	Affected Cloud Service	Mitigated
1	CVE-2015-1195	Image Service (Glance)	Yes
2	CVE-2015-1850	Volume Service (Cinder)	Yes
3	CVE-2015-1851	Volume Service (Cinder)	Yes
4	CVE-2015-5163	Image Service (Glance)	Yes
5	CVE-2015-7548	Compute Service (Nova)	Yes
6	CVE-2015-3221	Network Service (Neutron)	No*

Table 1: Information flow control vulnerabilities in OpenStack.

To address this problem, we enhance libvirtd with an in-daemon security framework that can validate whether the requesting event handler has the same label as the resources to be operated upon, similar to SEPostgreSQL [36]. As shown in Figure 7, the basic idea is that when an event handler establishes a connection to libvirtd through a Unix domain socket, the in-daemon security framework retrieve labels of the event handler from the socket descriptor (e.g., via *getpeercon*). Then, when the event handler requests libvirtd for a VM operation, the security framework compares the label of the event handler with the label of the VM, ensuring that the event handler is authorized to operate over the VM. At present, this enforcement mechanism is embedded into the libvirtd. We will explore using the Pileus kernel as the security server in the future.

6. EVALUATION

6.1 Mitigating Cloud Service Vulnerabilities

In this section, we show the security improvement made by Pileus over the off-the-shelf OpenStack, both through a system exploit experiment and a qualitative analysis.

Exploit Experiment. We ported OpenStack Icehouse 2014.1 to Pileus. Six information flow vulnerabilities were reported after our installation. Five of them are present in our deployment and one is not. To conduct the comparison, we did not patch the cloud services and try to exploit them in vanilla OpenStack and OpenStack on Pileus respectively. The vulnerabilities are listed in Table 1.

Vulnerability 1 is a pathname resolution bug in image service. Exploiting the vulnerability, we were able to read arbitrary image files on an image node that runs vanilla OpenStack image service. In contrast, Pileus successfully prevented the vulnerable image service from reading other users' images since the event handler of image service is confined to a user label.

Vulnerability 2, 3, 4 and 5 are of a similar kind: by exploiting them, we were able to read/overwrite arbitrary files on a cloud node. The attacks happen due to a helper program `qemu-img` that was called by cloud services to process user images. If not explicitly specified, `qemu-img` will infer image type and automatically read necessary files (e.g., base file for a qcow2 type image) to build the image. An adversary may thus trick a vulnerable cloud service into accessing files that he does not have access. Although the vulnerabilities remain unpatched, Pileus successfully prevented vulnerable cloud services from being utilized to access arbitrary files. In Pileus, `qemu-img` program runs with a user label inherited from the event handler that invokes it. Consequently, the only files that it can access are the ones that have the same label.

Vulnerability 6 was identified in OpenStack network service

Op	boot	delete	resize	snapshot	migrate	vol-attach
#	10	7	8	5	8	6

Table 2: Maximum number of nodes that needs to be trusted when performing cloud operations.

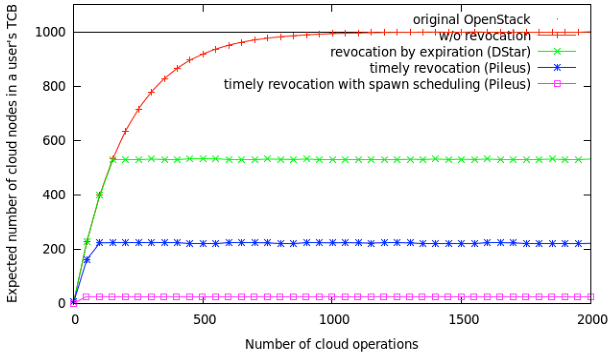


Figure 8: Expected number of cloud nodes in a user's TCB. The simulation consists of 1,000 cloud nodes. 5 are randomly picked each time to perform a cloud user's operation, and 10 operations are performed per second.

(Neutron) but not in the legacy nova-network that we used in our deployment. Thus we did not test against it. However, we note that by design Pileus can mitigate this vulnerability. The vulnerability is caused by incorrectly parsing iptables firewall rules that an adversary may leverage to block network connections of others. In Pileus, we used network namespaces to isolate the firewall rules for different users. Thus incorrect parsing of firewall rules can only affect a single network namespace, the adversary's own namespace.

Qualitative Analysis. In order to have a big picture of how Pileus can improve OpenStack security, we performed a qualitative analysis of all 154 vulnerabilities identified in OpenStack so far[6]. We found that 1/3 (53 out of 154) of OpenStack vulnerabilities are related to information flow problems studied in this paper, and Pileus systematically mitigate those vulnerabilities.

6.2 Reducing the Cloud Users' TCBs

In the original OpenStack, a user needs to rely on all cloud nodes to execute their user operations securely, so all cloud nodes are in the trusted computing base (TCB). Consequently, a compromise of any single cloud node allows adversary to gain control over any user's data, cloud wide. In contrast, Pileus restricts the data accessibility of a cloud node to the authority held by it. Thus, data loss due to a node compromise is bounded by the trust placed on the node. Table 2 shows the maximum number of nodes that need to be trusted in order to perform various cloud operations on Pileus[7]. As shown in the table, the size of each operation's TCB is reduced to a handful of cloud nodes that are actually involved in each user's operation, instead of the entire cloud.

In addition, Pileus further reduces the amount of time that a user needs to trust a cloud node in an operation through its timely revocation mechanism. This reduces temporal attack surface of a user's TCB. To show the effect of the mechanism, we saturate a cloud with a large number of concurrent user operations and evaluate the average size of the user's TCB[8]. Figure 8 shows the result. The simulated cloud has 1,000 cloud nodes, and to simplify the discussion, we ignore the actual service deployment and assume each user operation takes five cloud nodes picked at random. We then investigate the effectiveness of different approaches by comparing the expected number of cloud nodes of a user's TCB (Y-axis in the figure). The X-axis is the number of cloud operations performed by the user.

In original OpenStack, a user's TCB includes all the cloud nodes (black line at top). In contrast, when enforcing a decentralized security principle, a user's TCB dynamically expands as more cloud nodes are involved in his operations (red line). However, without revocation, the user will eventually end up trusting all the cloud nodes in the cloud. When an expiration-based strategy is adopted, the user's trust is revoked from a cloud node after a certain period. In our simulation, we set the expiration time to be 15 seconds. The expected size of the user's TCB in this case converges to around 530 cloud nodes (green line). In contrast, Pileus adopts a timely revocation where trust is revoked immediately after a cloud node completes its processing of the user's operation. In the experiment, we used our observed service duration of 2 to 8 seconds in a uniform distribution. In this case, the expected size of the user's TCB converges to around 220 cloud nodes (blue line). When we adopt Pileus's spawn scheduling algorithm, which gives priority to cloud nodes that are already within a user's TCB, the size of user's TCB converges to 25 cloud nodes (pink line) in this experiment, assuming that each cloud node can serve at most 10 concurrent operations.

6.3 Optimizing the Cloud Users' TCBs

Pileus differs from previous DIFC approaches is its ability to dynamically manage TCB on behalf of cloud users. Its ownership registry runs a spawn scheduling algorithm that computes a spawn destination, implementing a best effort approach to reduce the likelihood of compromise of a user's TCB. To approximate the likelihood of a user's TCB compromise, we propose a metric called *Averaged TCB Sharing Factor (ATSF)*. The ATSF metric reflects an intuitive observation—the more a user's TCB overlaps with other users, the more likely it is to be compromised by one of those other users (assuming users are equally likely to be an adversary). It is calculated based on the average number of users per cloud node, using the following equation where U_i is the number of unique users on cloud node i. In an ideal case where each user has nooverlapping TCB, ATSF would be one. We thus measure three different node selection strategies in OpenStack, and compare them with Pileus's spawn scheduling algorithm.

$$\frac{\sum_{i=1}^{n} U_i}{\sum_{i=1}^{n} N_i} \quad where \quad N_i = \begin{cases} 0, & U_i = 0 \\ 1, & U_i > 0 \end{cases} \tag{6.1}$$

The result is shown in Figure 9. The simulation consists of 1,000 cloud nodes and 400 cloud users. Each cloud node has a capacity of 10 operations, i.e., a node can support a maximum of 10 concurrent cloud operations. The X-axis is the total number of cloud operations performed. They are randomly distributed across 400 cloud users. The Y-axis is measured ATSF. The higher the ATSF is, the more TCB sharing is observed across cloud users. Thus the TCB of a cloud user is more likely to be compromised.

The first node selection strategy in OpenStack is maximum utilization. This strategy tries to maximize the utilization of individual cloud nodes, i.e., unless a cloud node reaches its capacity, it will be scheduled first. Such strategy is useful when cloud vendor wants to minimize its cost (e.g., electric bill). As shown in the figure, the ATSF quickly reaches 10 (the capacity of a cloud node) after

[6] These vulnerabilities came from OpenStack versions spanning from 2012 to 2016. Therefore much of this evaluation is necessarily qualitative.

[7] The maximum occurs when every event handler involved in the operation runs on its own cloud node. The actual number is often much smaller since Pileus will always try to schedule event handlers of the same cloud operation on the same cloud node.

[8] In this case, a user's TCB at a given time becomes a composite of all cloud nodes that are involved in concurrent operations.

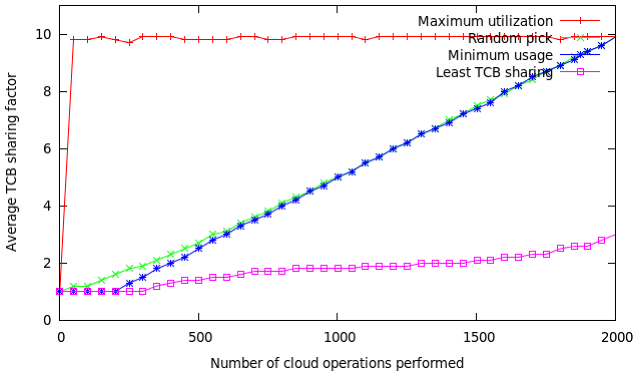


Figure 9: Average TCB sharing factor under different node selection strategies. The simulation consists of 1,000 cloud nodes and 400 cloud users. Cloud operations are randomly distributed across cloud users.

Type	Number	Example
DIFC-aware	13	nova boot
DIFC-unware	135	nova volume-attach
Infrastructure	6	nova host-action
Multiple users	3	nova host-evacuate
Total	157	

Table 3: OpenStack operations.

around 10 cloud operations and stays at 10 thereafter. The second node selection strategy randomly selects cloud nodes for a user's operation. The ATSF increases almost linearly as the number of operations grow. The third node selection strategy selects least occupied cloud nodes, i.e., cloud nodes currently performing least cloud operations. Such strategy is useful when cloud vendor wants to improve the performance of individual cloud operations. For this strategy, the ATSF stays at 1 before all the 1,000 cloud nodes are used by cloud users. Then the ATSF increases linearly, at a rate almost the same as the random selection strategy. The ATSF for both the random selection and least usage strategy eventually reaches 10, when the cloud is saturated with operations (at ∼2000 operations). In contrast, Pileus's spawn scheduling algorithm tries to co-locate the same user's cloud operations on the same cloud nodes, thereby reducing the ATSF. As shown in the figure, the ATSF is 1 before all 1,000 cloud nodes are used. Then it increases slowly and reaches a maximum of around 3 when the cloud is saturated with cloud operations. This simulation shows that Pileus's spawn scheduling algorithm indeed reduces TCB sharing among users.

6.4 OpenStack on Pileus

OpenStack Operations. One concern of DIFC approach is that cloud services need to be intrusively modified to be aware of DIFC control. However, as we show in this section, majority of cloud operations do not require cloud services to be DIFC aware. Table 3 shows cloud operations we studied. Out of the 157 cloud operations that are available in our deployment, 135 do not require any DIFC aware cloud services to run. What this means is that despite of being confined, cloud services involved in these operations are not aware of Pileus. 13 operations may require certain cloud services to be DIFC aware—they need to declassify or endorse data on behalf of cloud users. But as we show later, the types of endorsement and declassification are limited.

The remaining nine cloud operations are special cloud administrator operations that fall into two categories. The first category allows cloud administrator to directly operate over the cloud infrastructure. These operations do not involve information flow of user, so Pileus confines cloud services to a vendor label. The second category involves operations that operate over multiple users' data at the same time (e.g., delete all VMs on a node). In this case, Pileus confines cloud services using group label, which combines authorities of multiple users that are involved in the operation.

Syscalls (μs)	Native Linux	Pileus	Pileus Mult.	FlowK Mult.	Flume Multi.
open					
—create	1.23	6.16	5	8.3	16
—exists	0.62	2.90	4.7	11	34.5
—not exist	0.51	1.69	3.3	3.7	23.6
close	0.51	0.55	1.1	1.1	1.3
stat	0.33	1.55	4.7	N/A	34.5
readlink	0.34	1.52	4.5	N/A	33
unlink	11.97	24.31	2	N/A	7.2
fork+exit	263.4	287.5	1.1	N/A	N/A

Table 4: System call overheads compared with Flume [21] and FlowK [48]. Results are averaged over 10,000 runs.

Endorsers and Declassifiers. A cloud operation would involve DIFC-aware cloud services only if it will cause data to flow across user boundaries (i.e., resource sharing). In our study, we found that only two kinds of resources, *volume* and *image*, might be shared across users. Although 13 cloud operations involve volume or image sharing, the types of declassification and endorsement are limited, as shown below.

User → Public declassification allows a user to safely release a private resource to public. For images, the declassifier is motivated by the problem studied in Amazonia [8], where a careless user may publish her images with sensitive data such as API keys remained. We thus implemented the countermeasures suggested in Amazonia which parses the image and scans for any sensitive data. The image declassifier achieves the following security guarantee: without proper declassification, user images will not be released to public either intentionally or by mistake.

For volumes, declassification to public means removing any data residue. So, to implement a volume declassifier, we factored out the function in OpenStack volume service that zeros out all data on a volume. The difference is that in OpenStack, vulnerabilities [12, 13] can cause this function to be omitted. But on Pileus, the volume declassifier must be run to return an used volume.

Public → User endorsement allows a user to safely use a public resource. For images, the problem is motivated by vulnerabilities such as [41]. We provided two reference implementations. One endorses image by performing a checksum against a white list and another scans the image for malware.

User → User data flow allows a user to selectively share a resource with another user (e.g., Alice shares her volume with Bob). In this case, the owner of the resource needs to declassify the data (i.e., run a declassifier), and the receiver needs to endorse the data (i.e., run an endorser). To implement a declassifer for the owner is different from declassifying to the public, since in this case the resource may contain private data that the owner wants only the receiver (e.g., Bob, not public) to be able to access. The general idea is that declassifier will still run with the owner's ownership, as it is trusted by the owner to declassify data, but it will create a new intermediate secrecy tag, say n, to label the resource and transfer the ownership of n to the receiver. Then using the ownership of n, the receiver, and only the receiver, will be able to access the data. The receiver may endorse the resource using the same endorsers as if the resource is from public.

Latency(s)	boot	delete	resize	snapshot	migrate
OpenStack	3.92	1.72	4.07	2.83	4.11
Pileus	4.01	1.74	4.16	2.91	4.20
Percentage	2.3%	0.8%	2.2%	2.8%	2.2%

Table 5: Latency for cloud operations.

6.5 Performance

Our testbed consists of six cloud nodes: three running nova-compute, one running Glance services, one running Cinder services and the last one running the rest. Cloud nodes are identical blades with 2.4Ghz Intel E5-2609 CPU and 64GB memory, installed with Ubuntu 14.04.

Pileus Kernel. Table 4 shows micro-benchmark results for some system calls. The process under test has secrecy and integrity label with both 20 tags. For most system calls, Pileus kernel adds a latency of a factor of 1.1-5 with relative to native Linux. Since Pileus runs the DIFC mechanism in kernel, the performance is better than Flume [21]. It also appears to be slightly better than FlowK [48], another kernel DIFC module developed contemporarily.

Pileus Daemon. We evaluate the throughput of Pileus daemon by stressing it with large volume of events. In original OpenStack, the throughput for cloud service is 1,200 req/sec whereas in Pileus it is 950 req/sec (20.8% slowdown). The reason is that cloud services spawn green threads but Pileus Daemon spawns processes, in order to isolate them using user labels. To improve the performance, a possible solution is to build new OS abstractions that can be as lightweight as threads but have the same level of isolation as processes, such as the *event process* abstraction proposed in Asbestos [15]. But this may require intrusive kernel modification.

Ownership Registry. We evaluate the scalability of the OR by stressing it with high frequency of spawn request. Results show that the OR can handle up to ∼3000 req/sec. Most of the overhead comes from two sources: (1) the spawn scheduling algorithm and (2) the OR signing the authority token during spawn. One way to optimize it is to separate the spawn scheduling algorithm into a separate service. This service needs not to be trusted, but the OR must be able to check the output of the service to ensure that the global cloud policy is met (e.g., CoI is not violated).

Overall Latency. Table 5 shows the latency perceived by cloud users. While Pileus adds latency in its network protocol (due to the added round of communication with the OR during spawn), the latency is amortized by the time spent on actually processing the events. As a result, we noticed less than 3% additional latency when performing various cloud operations in Pileus.

7. RELATED WORK

There has been much work on improving data security in cloud, including data encryption [31], data sealing [34], protection against compromised hypervisor and privileged domain [53, 9, 6, 46] and leakage detection [30]. These works aim to protect data from parties in cloud that should not have access. Pileus addresses a different concern: if a cloud service has legitimate access to data, how to prevent them from being leveraged as confused deputies due to cloud service vulnerabilities.

Another line of research focuses on security of cloud infrastructure. The CV framework [35] allows cloud users to reason about integrity of cloud nodes. CloudArmor [44] protects cloud operations performed on benign cloud nodes from compromised ones by enforcing a cloud operation model. These works are complementary to Pileus. SOS [45] addresses the concern of compromised compute services, but it requires other cloud services to be trustworthy. In contrast, Pileus can systematically run and confine any type of cloud services. Pileus is motivated by the SCOS [43], which advocates the development of a secure cloud operating system in order to confine vulnerable cloud services.

Pileus takes advantage of a number of well-established mechanisms in decentralized information flow control (DIFC) systems. In particular, Pileus adopts its label and ownership from Flume [21] and its event handler abstraction from Asbestos [15]. The DIFC model was proposed by Myers and Liskov [24], then it was incorporated into programming languages such as Jif [25] and systems such as Asbestos [15], HiStar [51], Flume [21], Laminar [33] and Aeolus [11]. These systems often assume a fully trusted reference monitor (or several mutually trusted reference monitors) that can track information flows on the system. Pileus, on the other hand, assumes cloud nodes are mutually distrustful.

Similar to DStar [52], Fabric [23] and Mobile-fabric [4], Pileus assumes reference monitor on a single node may be compromised and therefore cloud nodes are mutually distrustful. However, Pileus differs from these systems in its ability to control authority distribution. In Pileus, the ownership registry (OR) ensures that authority propagation across cloud nodes will not violate the cloud policy and it enables timely authority revocation from nodes. In addition, Pileus developed a systematic approach for cloud users to delegate their authorities to event handlers that they trust, without the fear the such trust might be misused to run other code.

Researchers have shown that DIFC is an useful model in protecting distributed web applications deployed on PaaS clouds [29, 5]. These systems focus on protecting cloud hosted applications and rely on a trustworthy cloud platform, including underlying cloud services and nodes. Pileus can be used to secure this foundation.

The ownership authorization in Pileus is motivated by capability-based systems [37, 47, 16]. A security issue with traditional capability systems is that they cannot enforce the ⋆-property [49]. To address this concern, multiple designs [20, 18, 16] were proposed that combined capability with authority check to limit who may exercise the capability. Pileus adopts a similar design idea that uses ownership authorization to securely delegate the ownerships to particular event handlers that cloud users trust.

Cloud vendors have started developing some countermeasures to address the security issue reported in this paper. For example, an OpenStack blueprint [28] proposes to encrypt the message between cloud services, preventing a malicious cloud node from sniffing the channel. As another example, OpenStack supports scoped tokens [2] which could reduce the token privilege down a single project. However, much of these efforts are still ongoing, and they cannot address the vulnerabilities in cloud services.

8. CONCLUSION

Pileus is a model and system for securing cloud platforms by enforcing decentralized information flow control (DIFC) over cloud services. On Pileus, cloud services are ephemeral, and are confined to users' security labels, enabling least privilege. Pileus tracks and protects users' data as it flows through the cloud platform therefore mitigating both cloud service vulnerabilities and compromised cloud nodes. We ported OpenStack, a widely used cloud platform, to Pileus and show that Pileus can greatly improve the security of OpenStack for less than 3% overhead on user operation latency.

9. ACKNOWLEDGMENTS

The authors thank Danfeng Zhang, Susanta Nanda and the anonymous reviewers for their comments on drafts of this paper. This work was supported by NSF under grant No. CNS-1117692.

10. REFERENCES

[1] CVE-2012-3360. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3360.

[2] Openstack keystone token. http://docs.openstack.org/admin-guide/keystone_tokens.html.

[3] Amazon EC2. http://aws.amazon.com/ec2.

[4] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *Proc. 2012 IEEE Security and Privacy*, 2012.

[5] J. Bacon, D. Eyers, T. Pasquier, J. Singh, I. Papagiannis, and P. Pietzuch. Information Flow Control for Secure Cloud Computing. *IEEE Transactions on Network and System Management, SI Cloud Service Management*, 11(1):76–89, 2014.

[6] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proc. 11th USENIX OSDI*, 2014.

[7] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.

[8] S. Bugiel, S. Nürnberger, T. Pöppelmann, A. Sadeghi, and T. Schneider. AmazonIA: When elasticity snaps back. In *Proc. ACM CCS'11*.

[9] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy. Self-service cloud computing. In *Proc. ACM CCS'12*.

[10] CVE-2012-0030. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0030.

[11] W. Cheng, D. R. K. Ports, D. A. Schultz, V. Popic, A. Blankstein, J. A. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for usable information flow control in aeolus. In *USENIX ATC'12*.

[12] CVE-2012-5625. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-5625.

[13] CVE-2013-4183. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4184.

[14] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[15] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *Proc. ACM SOSP'05*.

[16] L. Gong. A secure identity-based capability system. In *Proc. IEEE Security and Privacy*, 1989.

[17] N. Hardy. The confused deputy. *Operating Systems Review*, 22(4):36–38, Oct. 1988.

[18] P. A. Karger. Limiting the damage potential of discretionary trojan horses. In *Proc. IEEE Security and Privacy*, 1987.

[19] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 2–12, 1984.

[20] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proc. IEEE Security and Privacy*, 1984.

[21] M. N. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. ACM SOSP'07*.

[22] libselinux. http://www.rpmfind.net//linux/RPM/fedora/devel/rawhide/armhfp/l/libselinux-2.4-5.fc24.armv7hl.html.

[23] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. ACM SOSP'09*.

[24] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. 16th ACM SOSP*, 1997.

[25] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM TOCS*, 9(4):410–442, Oct. 2000.

[26] Security-enhanced linux. http://www.nsa.gov/selinux.

[27] OpenStack Open Source Cloud Computing Software. http://www.openstack.org//, 2008.

[28] OpenStack Message Security. https://wiki.openstack.org/wiki/MessageSecurity/.

[29] T. Pasquier, J. Singh, D. Eyers, and J. Bacon. CamFlow: Managed Data-Sharing for Cloud Services. *IEEE Transactions on Cloud Computing*, 2015.

[30] C. Priebe, D. Muthukumaran, D. O' Keeffe, D. Eyers, B. Shand, R. Kapitza, and P. Pietzuch. Cloudsafetynet: Detecting data leakage between cloud tenants. In *Proc. ACM CCSW'14*.

[31] K. P. N. Puttaswamy, C. Kruegel, and B. Y. Zhao. Silverline: Toward data confidentiality in storage-intensive cloud applications. In *Proc. 2nd ACM SOCC*, 2011.

[32] Rackspace Cloud Servers. http://www.rackspace.com/cloud/.

[33] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proc. ACM PLDI*, 2009.

[34] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Proc. 21st USENIX Security*, 2012.

[35] J. Schiffman, Y. Sun, H. Vijayakumar, and T. Jaeger. Cloud verifier: Verifiable auditing service for IaaS clouds. In *Proc. IEEE SERVICE'13*.

[36] The SEPostgreSQL Project. https://wiki.postgresql.org/wiki/Main_Page.

[37] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: A fast capability system. In *Proc. ACM SOSP'99*.

[38] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 166–176, 2000.

[39] CVE-2012-4573. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4573.

[40] CVE-2012-5482. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-5482.

[41] CVE-2013-4354. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4354.

[42] CVE-2015-3221. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3221.

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

[44] Y. Sun, G. Petracca, T. Jaeger, H. Vijayakumar, and J. Schiffman. Cloudarmor: Protecting cloud commands from compromised cloud services. In *Proc. IEEE CLOUD'15*.

[45] W.-K. Sze, A. Srivastava, and R. Sekar. Hardening OpenStack Cloud Platforms against Compute Node Compromises. Technical report, ASIACCS 2016, May 2016.

[46] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: making trust between applications and operating systems configurable. In *Proc. USENIX OSDI'07*.

[47] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In

Proc. ICDCS'86.

[48] D. M. E. Thomas F. J.-M. Pasquier, Jean Bacon. Flowk: Information flow control for the cloud. In *Proc. IEEE CloudCom'14*.

[49] W.E.Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proc. 7th DoD/NBS Computer Security Conference*, 1984.

[50] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium*, volume 8032, page 6, 2002.

[51] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. USENIX OSDI'06*.

[52] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proc. USENIX NSDI'08*.

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