On the Universally Composable Security of OpenStack[††]

Kyle Hogan[*], Hoda Maleki[†], Reza Rahaeimehr[‡], Ran Canetti[§¶], Marten van Dijk[‡],
Jason Hennessey[§‖], Mayank Varia[§], Haibin Zhang[**]
[*]Massachusetts Institute of Technology. klhogan@mit.edu.
[†]Augusta University. hmaleki@augusta.edu.
[‡]University of Connecticut. reza.rahaeimehr, marten.van_dijk@uconn.edu.
[§]Boston University. canetti, henn, varia@bu.edu.
[¶]Tel Aviv University.
[‖]NetApp.
[**]University of Maryland, Baltimore County. hbzhang@umbc.edu.

***Abstract*—We initiate an effort to provide a rigorous, holistic and modular security analysis of OpenStack. OpenStack is the prevalent open-source, non-proprietary package for managing cloud services and data centers. It is highly complex and consists of multiple inter-related components which are developed by separate, loosely coordinated groups. All of these properties make the security analysis of OpenStack both a worthy mission and a challenging one. We base our modeling and security analysis in the universally composable (UC) security framework. This allows specifying and proving security in a modular way — a crucial feature when analyzing systems of such magnitude. Our analysis has the following key features:**

1) **It is *user-centric*: It stresses the security guarantees given to users of the system in terms of privacy, correctness, and timeliness of the services.**
2) **It considers the security of OpenStack even when some of the components are compromised. This departs from the traditional design approach of OpenStack, which assumes that all services are fully trusted.**
3) **It is *modular*: It formulates security properties for individual components and uses them to prove security properties of the overall system.**

Specifically, this work concentrates on the high-level structure of OpenStack, leaving the further formalization and more detailed analysis of specific OpenStack services to future work. Specifically, we formulate ideal functionalities that correspond to some of the core OpenStack modules, and then proves security of the overall OpenStack protocol given the ideal components.

As demonstrated within, the main challenge in the high-level design is to provide adequately fine-grained scoping of permissions to access dynamically changing system resources. We demonstrate security issues with current mechanisms in case of failure of some components, propose alternative mechanisms, *and rigorously prove adequacy of then new mechanisms within our modeling.*

***Index Terms*—Modular Security Analysis, Universal Composability, Cloud Security, OpenStack**

I. Introduction

OpenStack is a software package for data centers and virtualization services, including remote computation, storage, networking, and related services. The OpenStack project began in 2010 as a collaboration between two groups: Rackspace, a public IaaS provider selling cloud services, and NASA, a part of the United States government that wanted to take advantage of the elasticity and datacenter efficiency benefits that come from combining different workloads into a single private cloud [1]. Since then, it has grown to be a large open source project with over 9 million lines of code, and over 6,000 contributors and hundreds of implementations around the world [2], [3]. There is a governance body [4] that actively manages the development and stability of OpenStack. Analyzing security properties of such a large-scale information system is a daunting task.

A first challenge is to adequately articulate and rigorously express the security requirements of the system in the first place. Indeed, adequately capturing even simple, intuitive concerns is non-trivial. Furthermore, security is often inseparable from the expected functionality, which is complex in and of itself. It also invariably has multiple facets and competing requirements that need to be reconciled.

A second challenge is to rigorously assert the specified properties. This challenge is even more daunting, especially when the system consists of multiple components and one has to take into account inter-component interactions, potential failure of individual components, and the associated potential vulnerabilities.

Modular security analysis: The natural way to deal with such complexities is modularity: formulate and assert the security properties of individual components, and then deduce security properties of the overall, composite system from the security properties of the components, as well as those of the way in which the components are put together. This breaks down the overall analysis into multiple steps where each step deals with a much simpler system. Furthermore, when successful, the analysis would deduce the overall security of the system from the security of the components, and the security of the overall design given the components. Still, breaking down a system to components in a way that allows for effective composable *security* analysis is a non-trivial task

[††] The first three authors contributed equally to this effort.

in and of itself.

A number of frameworks for modular security analysis for cryptographic protocols have been developed over the years, e.g. [5]–[12]. Furthermore, a number of works have used these frameworks to analyze security of security-sensitive systems that are non-cryptographic in nature, e.g. [13]–[16]. Extending these cryptographic frameworks to handle systems with complex interfaces and sizeable codebase is a challenging endeavor — but one that holds great promise. In particular, it opens the door to a rigorous, yet modular and approachable security analysis of large-scale software systems. Indeed, such analysis can be very valuable even in systems that use little or no cryptography.

Our contributions: In this work, we initiate modular security analysis of OpenStack, which is a large-scale distributed system with complex interfaces whose security is important to many cloud computing use cases. We perform our analysis within the Universally Composable (UC) security framework, which provides a way to articulate security properties rigorously and precisely — and supports security-preserving modularity.

We first formulate specifications (in form of ideal funcrionalities within the UC framework) that capture the overall functionality and security requirements of the openstack suite, from the point of view of an outside user of the system.

We then formulate specifications (in form of ideal funcrionalities within the UC framework) that capture the functionality and security requirements of a selected set of OpenStack services, specifically Keystone (key registry), Nova (compute), Glance (image repository), Cinder (storage), Horizon (user dashboard).

We then investigate the OpenStack mechanism for realizing the overall specification (i.e. for realizing the overall ideal functionality) given ideal realizations of the said services.

Importantly, here we consider the realistic case where some of the services might be compromised or adversarially controlled, and the goal is to keep providing security for the non-compromised mechanisms.

It turns out that the main challenge in this overall protocol is to provide a sufficiently nimble, fine-grained and flexible mechanism for controlling access to various services that change. Indeed, we demonstrate that prevalent protocols (e.g. the bearer token mechanism) have security flaws.

We then propose a new token mechanism and demonstrate that, with this mechanism in place, the overall protocol indeed UC-realizes the overall OpenStack specification, given the ideal functionalities representing the services.

In summary, this work makes the following contributions.

- Demonstrate the viability of Universally Composable (UC) security analyses for a system with the complexity of OpenStack.
- Describe the interaction between the main OpenStack components, while abstracting the $\sim$9 million of lines of code that collectively realize these components.
- Analyze the composite security provided collectively by these OpenStack components, and how security of the whole is impacted if some of the parts are corrupted.
- Propose a specification for an improved token mechanism, along with an analysis of the concrete security improvements it offers to OpenStack.

We emphasize that in this work, we are not doing formal/symbolic/automated analysis. Clearly, it would be great if we could do the proofs within a pl-style formal framework. However, the main contribution here should be seen as not so much the proof, but the dissection of the problem and determining *'what'* it is that we want to prove (i.e. the functionalities). This is a task that cannot be mechanized with current technology, and this is where our main contribution lies.

Lessons Learned: The main challenge/goal of the high-level design is to provide a sufficiently nimble and fine-grained access control mechanism where users will be able to create and configure tasks that involve multiple services and at the same time, keep rogue entities from gaining undesired access. This is further complicated by the fact that users may not be aware of all the services they are using and so, the services themselves should be able to act on behalf of users. We investigate the current mechanism, which is based on a token that contains the access control information for the task at hand and what is being passed in all communications. We point out how some design flaws are manifested in the current formalism and then, propose and *analyze* an alternative token mechanism. The UC framework is critical in several levels: (a) it allowed us to identify the above issue (it is hard to lose sight of it when dealing with all details at once). (b) it allowed us to define security of the mechanism, and rigorously assert that the proposed mechanism satisfies the definition.

Security weaknesses formalized and contextualized: As mentioned above we have learned the main security challenge in a multi-tenant data center management system like OpenStack is to provide an appropriately fine-grained, yet secure mechanism for controlling the access of users (and agents of these users within the system) to services. This includes controlling access to compute, storage, networking, and other services, as well as preventing users from stepping into each others data and virtual machines. Indeed, much of our attention has been focused on analyzing the mechanisms provided by OpenStack to provide this control.

The design of OpenStack implicitly assumes that all components of an OpenStack-based service are trusted. Our work demonstrates that as long as this assumption holds, the OpenStack design indeed provides adequate security: namely, secrecy and correctness of data and computations. Conversely, our analysis formally shows the extent to which OpenStack is vulnerable (and also the extent to which it remains safe) when a subset of components is compromised.

We remark that the case where some components become compromised is quite realistic. Indeed, one known security concern within OpenStack is that the VM manager Nova is more susceptible to attack than other services because it is exposed to a richer attack surface from malicious VMs. Specifically, if an attacker is able to compromise only one VM by exploiting any vulnerability, then in fact he can compromise the compute node hosting the VM and get the credential of the compute-node. By having the credential of one compute-node, the attacker can observe or even modify all the messages in Nova message queue, including all tokens passed to Nova. This realistic example shows that it is prudent to design the system so as to minimize the damage from the compromise of individual components, and to perform analysis that provides some security guarantees even in case that some components are adversarially controlled.

We turn to describing the flaw in *bearer token* mechanism to authenticate users and verify their authorization to access resources. Bearer tokens given by OpenStack's credentialing service Keystone effectively permit a user to pass her credentials to services that can then make actions on her behalf. As long as the communication between services is secured via point-to-point secure session protocols (say, via TLS), this mechanism provides security against external attackers that only control the network. However, this mechanism allows a corrupted component (say, Nova) to impersonate tokens on behalf of any user. When the inter-process communication is not secured in a point-to-point way, *any* rogue OpenStack entity that can eavesdrop to the inter-service communication (say, a hypervisor that was compromised by its tenant VM) can potentially have access to all current bearer tokens in the system. Indeed, previous works (e.g. [17]) have already pointed out this weakness and proposed limiting the scope of these bearer tokens by setting expiration times and other scoping mechanisms.[1] It should be stressed that, upon each new use of the token, each new service verifies the token again with Keystone. Ergo, tokens that are invalid will not cause damage. However, when the tokens are broadly scoped, nothing prevents a rogue component from using legitimate tokens of existing unsuspecting users to compromise both the integrity and the secrecy of their data.

This work analyses OpenStack with two token mechanisms. First, we analyze OpenStack's existing bearer tokens. Our analysis formulates that the current OpenStack realizes an "ideal cloud" specification that provides little security as soon as any component is corrupted. We then specify the attributes of a stronger *one-time* token mechanism; we show that the additional security provided by the limitation to one-time use, together with the ability to identify the entity that provides the token, suffices for realizing a significantly stronger variant of the ideal cloud specification that limits the damage caused by corrupted services.

[1]For simplicity of exposition we leave the timeout mechanism (as well as measurement of real time) outside the model. We note that timing mechanisms can be added in a relatively straightforward way, using the UC-style modeling of network time of Canetti et al. [18]. Indeed, the ability to modularly add the consideration of time is another demonstration of the power of composable security analysis.

Towards modular and mechanized analysis: One of the most important aspects of this work, that sets it apart from many previous works in the UC framework, is that we provide in full detail the specifications of the ideal cloud and the individual services, as well as the descriptions of the simulators and the proofs of security, without glossing over steps. As a consequence, our proofs and specifications are decently long and tedious. Indeed, while for this paper we stick to pen-and-paper proofs, we believe that our modeling and analysis are readily amenable to mechanization, and also to some level of automation. Natural candidates for tools that would enable such mechanized analysis include the EasyCrypt tool [19], the FCF tool [20], or the CryptHOL tool [21].

Organization: We begin in Section II by providing some background for OpenStack. Section III points to some related works and Section IV describes our approach toward the security analysis of OpenStack. Section V provides an informal account of our modeling of OpenStack services and the security properties we chose to model. We also motivate our design decisions. We then proceed to introduce the modeling in more detail. Specifically, Section VI presents the ideal cloud, namely our security specification. Section VII presents our modeling of selected OpenStack services. In Section VIII we explain the properties needed for stronger tokens (i.e., more secure) variant of the ideal cloud. We conclude by discussing future work in IX.

II. Background

Due to space constraints, we relegate an overview of universally composable (UC) security to the full version of this paper [22], and also refer interested readers to [7] and [23] for more details.

In this section, we focus on surveying OpenStack. As outlined in the Introduction, OpenStack is a modular, distributed, open-source cloud computing software stack for providing Infrastructure as a Service (IaaS) to multiple (potentially untrusting) users. In this section, we describe OpenStacks operation with a focus on some of the security concerns of its authorization system.

1) Modular services: The design of OpenStack is inherently modular, with 23 modules where each module has some pre-specified functionality, as well as interfaces with the other modules it interacts with. It should be noted though that the functionality of the interfaces is not completely pinned down; indeed some modules have multiple implementations that provide slightly different functionality. Also many of the modules allow for a variety of underlying software packages as plug-ins. Some of the main modules of OpenStack include:

Nova (compute): Manages the creation, maintenance and removal of virtual machines (VMs).

Glance (image repository): Stores and manages the images loaded to VMs.

Cinder & Swift (block & object storage): Manage the storage of data (in blocks, volumes, and more general objects) for VMs.

Neutron (networking): Provides internal and external virtual networks for VMs.

Keystone (access control and key management): Holds the permission information controlling the access of users to the services and data. Interacts with users and all other modules to enforce the permission policies.

Horizon (user-side dashboard): Provides an interface between users of the system and its service modules.

Each one of these modules is a complex, distributed system in and of itself, sometimes with multiple subdivisions, plugins, and alternative implementations.

2) Trust model: Many of OpenStack's design choices and security issues stem from its broad trust model, which assumes that all services act as faithful user agents. Providing security even in the case where some services are comporomized does not appear to be a design goal. Furthermore, interactions between services in OpenStack are optimized in light of this trust. However, OpenStack's unprotected interior means that a (partially) compromised service can do a great deal of harm: acquiring a single bearer token allows the compromised service to impersonate the user for any subsequent action.

3) Tokens: To determine a user's project and role, Keystone gives the user a *bearer token* after authenticating with their credentials (e.g., username and password); users include this token in API requests (e.g., to create a new VM) to other services for authentication and authorization. Services pass this token to Keystone, which returns back a *(project,role)* tuple if the token was valid. Services then make all authorization decisions based on that tuple.

Importantly, services can continue to use the bearer token to make additional API requests of other services as necessary. For instance, the compute service is able to send a user's token to the storage service which can in turn verify with the identity service that this token has access to the requested volume and attach it to a node without needing to check with the end user itself. (A natural alternative to the token mechanism is a digital signature by Keystone regarding the user's capabilities. However, this solution has been rejected by the OpenStack community due to its computational and bandwidth overhead.) Bearer tokens are used similarly in other popular protocols, like OAuth [24]. Because possession of a bearer tokens grants access to resources, data in transit protection via TLS [25] is essential to protect the tokens from being viewed by unauthorized parties.

OpenStack uses plugins to Keystone to implement tokens, the most popular being UUID and Fernet [26]. UUID tokens issue random, universally unique identifiers [27] to users after a successful first authentication with Keystone, and stores them in a database with other required information such as expiration time, the project and role associated with it.

The Fernet token is a recent innovation that uses cryptography to provide authenticity without accessing a central DB. It is a mechanism by which keystone creates a private, authenticated channel to itself. It has quickly become the preferred token format for OpenStack as they do not require maintaining a central database of valid tokens, which adds network load and latency.

III. Related Work

A. Security Analysis of Clouds

The OpenStack Security Guide [25] goes into depth about the security of different aspects of configuring the many different pieces of OpenStack. However, it does not provide any security analysis, formal or otherwise, nor does it consider situations where a cloud service is compromised.

Other works have focused on the compromise of compute nodes [28] or parts of the management infrastructure [29], and Sun et al. [30]–[32] specifically discuss limiting the scope of compromised OpenStack services. These works conclude that corrupted cloud components have far-reaching security impact and can in many cases compromise the privacy and integrity of all cloud operations. Their conclusions highlight the need for a formal security analysis of service corruptions in OpenStack, which we provide with our construction. Sze et al. [28] additionally propose an alternative authorization tokening mechanism to reduce the effect of corrupted compute nodes, but their construction neither protects against compromise of other services nor provides token authentication and replay prevention. We have focused on addressing these requirements as well as shifting control of token generation and scope to the user responsible for the request. Also, crucially, we provide a security analysis that concretely specifies the security gain.

B. Using UC

Canetti et al. [33] show how the UC framework can be used to analyze the simple components of a file system in isolation and to guarantee that these components maintain their behavior in the larger system even under adversarial conditions. This demonstrates basic integrity properties of the file system, i.e., the binding of files to filenames and writing capabilities. Gajek et al. [34] evaluate in the UC framework the emulation of secure communication sessions by the composition of key exchange functionalities that are realized by the TLS handshake and record layer protocols. Canetti et al. [35] give a modular and global universally composable analytical framework for PKI-based message authentication and key exchange protocols.

For our analysis, we apply the style of [33] to the larger and more complex OpenStack framework and utilize aspects of [34], [35] to achieve secure communication. We further use our construction to demonstrate security flaws in OpenStack's current authorization mechanism and assess the improvements provided by our suggested changes.

C. Alternative Formalisms

The UC framework is not the only option for formal analysis of computing systems. In particular, Gu et al. [14] use the Coq proof assistant to analyze and provide an abstraction of layers of the computing stack including the kernel, networking, etc.

They developed and verified a certified kernel with 37 of these abstraction layers.

We chose to use UC for our analysis because its modularity and composability aligned well with the structure of OpenStack which is itself composed of many services that interact via a series of well defined APIs. These services support varying interchangeable implementations that would be difficult to support using a less modular proof framework.

IV. Our Approach

We initiate a study of the security properties provided by OpenStack when viewed as a service to external users which is a typical model for most (large scale) applications. This includes properties such as confidentiality and integrity of *data* (both in storage and in transition), confidentiality and correctness of *computations,* as well as timeliness and resource preservation. We also consider the extent to which these properties are preserved under various attack vectors and when various components of the system are compromised.

We base our analysis in the universally composable security (UC) framework, which provides a way to articulate security properties in a rigorous and precise way. According to the definition of universal composability, a UC-secure component remains secure if it is universally composed with other UC-secure components [7]. The extendability property of universal composability allows us to analyze a part of a system and additively analyze the remaining components. The framework provides a natural and convenient mechanism for arguing about the preservation of security when programs and systems are composed in a modular way. Indeed, from this perspective the UC framework appears to be ideally suited to analyzing OpenStack whose design is inherently and predominantly modular.

On the other hand, the UC framework was initially created, and predominantly used, for analyzing cryptographic protocols. These are very different than OpenStack: while their analysis requires creative reductions to hard computational problems, they are vastly simpler in terms of number of components, cases, and volume of code. Indeed, coming up with an effective modeling of OpenStack within the UC framework is a labor intensive, non-trivial line of research. This work paves the way in this direction.

Recall that in the UC framework the security requirements from the analyzed system (or, service) π are analyzed jointly with the functionality requirements from the service. This is done by way of formulating an *ideal service* $\mathcal{F}$, which specifies the desired response (or lack thereof) to any potential external input. Roughly speaking, the service π is said to emulate the ideal service $\mathcal{F}$ if no external environment can tell whether it is interacting with π or with $\mathcal{F}$.

In order to account for some level of allowable "slack" for π relative to $\mathcal{F}$, the framework allows the analyst to introduce an intermediary, or a *simulator* S that controls some of the interfaces between $\mathcal{F}$ and the environment. That is, service π is now said to emulate an ideal service $\mathcal{F}$ if there exists a simulator S such that no external environment can tell whether it is interacting with π or with a system where some of its APIs connect to $\mathcal{F}$, and other APIs connect to S. (Typically, S connects to APIs that we don't consider to be part of the desired functionality, such as the communication between components of the implementing protocol.)

An attractive property of this definitional style is the following natural security-preserving composability: Since the specification $\mathcal{F}$ is written as an "idealized" service in and of itself, one can design and analyze some other system (or, service) ρ where the components of ρ make calls to one or more instances of the service $\mathcal{F}$. The UC framework guarantees that the protocol $\rho^{\mathcal{F}\to\pi}$, where each instance of $\mathcal{F}$ is replaced by an instance of π, continues to exhibit the same security and correctness properties as the original protocol ρ. In particular, if ρ emulates some other ideal service $\mathcal{G}$, then $\rho^{\mathcal{F}\to\pi}$ will emulate $\mathcal{G}$ just the same. (Note that both π and ρ may well be distributed, multi-component systems in and of themselves.) Our goal is to demonstrate an approach that enables analysts to analyze the security of OpenStack in a structured and perceptible manner. To do so, we provide initial modeling and analysis of the overall design and operation of OpenStack, as well as the functionality and security requirements from a number of core modules (essentially the modules described above with the exception of Swift and Neutron). Our analysis validates the overall security of the design, while at the same time formulating some security weaknesses. Although, the weaknesses are conceptually known to the OpenStack community, our analysis shows the right level at which these issues must be dealt. For example, Sze et al. [28] tried to solve the token problem by assuming a trusted component inside Nova. Our analysis shows that such designs are not a suitable design decision if we are looking for a UC-secure system. We also propose and analyze methods for properly overcoming these weaknesses. Our analysis method developed in this paper covers the high-level design of some main components of OpenStack. While of course there are numerous vulnerabilities within each module that are beyond the scope of this foundational work, our method of analysis developed in this work paves the way for their eventual capture within the UC framework.

We first formulate an ideal cloud $\mathcal{F}_{Cloud}$ that provides a simple specification of the functionality and security that we assert OpenStack achieves. This formulation naturally involves many design choices and parameters that affect the security and functionality requirements imposed on the system. We discuss them within. One important aspect of our ideal cloud specification is the expected behavior upon various types of partial corruption (which correspond to corruption of individual modules in an OpenStack service). This is where we depart from the current OpenStack package, which does not provide any security guarantees as soon as any module is corrupted.

Next we formulate ideal functionalities that correspond to the four services we capture, namely $\mathcal{F}_{Compute}$, $\mathcal{F}_{Image}$, $\mathcal{F}_{BlockStorage}$, and $\mathcal{F}_{Identity}$. Our models for each OpenStack service aim at capturing the functionality and intricacies of the actual components of OpenStack, modulo some necessary modifications that are essential for security. Also here we face a number of choices that represent different levels of security of these services.

These services communicate with each other via secure message transmission $\mathcal{F}_{SMT}$. Additionally, they use an external network $\mathcal{F}_{ExtNet}$ to communicate with the user, or more specifically to connect to the user's *Dashboard* program (which is our abstraction of Horizon). Collectively, the joint interactive effort of these services and protocols comprise a cloud of *OpenStack Services*. In the two main results of our paper (Theorems 1 and 2), we prove that the OpenStack services collectively UC-realize our ideal cloud.

V. Modeling OpenStack Services

In this section, we provide an informal account of our modeling of OpenStack and the security guarantees we assert. We first describe the behavior of each service and the risk associated with its compromise. Then, we generalize from the service-level issues to provide informal, holistic security properties about OpenStack as a whole. Finally, we survey the design decisions and degrees of freedom that influence our model. The informal account in this section is then followed by the actual definitions of the OpenStack services (Section VII) and the ideal cloud (Section VI).

Following the approach of the UC framework, we consider an adversarial environment E that controls all the interfaces of the legitimate users with the analyzed service, and in addition controls the communication network and the compromised components of the system.

In the context of our OpenStack service, this means that E can create new compute nodes with specific images of its choice, and link nodes to storage volumes subject to their capabilities. In addition, E can delay or drop arbitrary traffic on the external network (e.g., the Internet) over which users communicate with OpenStack. Next, E can compromise one or more OpenStack services, and thus we reinforce the services to provide defense-in-depth against service-level compromise. We consider both passive corruptions in which the compromised services continue to function normally but only leak their internal states to E, and complete corruptions where the compromized services start running code provided by E.

It is stressed that, while the modeling and analysis considers only the interaction between E and a single instance of our cloud service, the universal composition theorem guarantees that the same security guarantees continue to hold even when E is interacting concurrently with other instances of our system and with arbitrary other systems.

A. Functionality and Security of Each Service

We begin by describing several functionalities that encapsulate both the functionality and security relationships between the OpenStack services and the user's dashboard protocol. In particular, we model the following functionalities in this work:

a) Dashboard: Unlike the services described below, the Dashboard protocol is owned and operated by a single user. The Dashboard specifies the sequence of service requests needed to satisfy the user's desires.

Compromising either the Dashboard or the user directly gives E the user's credentials. Hence, E can execute any operation that the user has privileges to perform, but cannot otherwise tamper with the services in any way; in particular, users never learn each other's credentials.

b) Identity: $\mathcal{F}_{Identity}$ is responsible for managing credentials. It communicates with all users and services. We presume that $\mathcal{F}_{Identity}$ is instantiated with credentials for each user and service; in practice, these credentials correspond to bearer tokens that can be acquired via an authentication protocol involving a username/password. Subsequently, when any service $\mathcal{F}_{Service}$ receives a request, it may ask $\mathcal{F}_{Identity}$ to validate whether the request is authorized based upon the credentials provided. Additionally, note that while OpenStack uses a project/role based permissions system, our modeling is agnostic to the design of credentials.

When E compromises $\mathcal{F}_{Identity}$ essentially has full control of OpenStack. It immediately acquires the credentials of all users and can even change the permissions associated with them. Furthermore, because all services outsource their authorization decisions to $\mathcal{F}_{Identity}$, E can make any request and convince all services to execute it.

c) Compute: $\mathcal{F}_{Compute}$ is responsible for managing the computing nodes on the cloud. It expects that the commands it receives over the network originate with the user's dashboard service. Then, it relies upon the other OpenStack services to aid in fulfilling these requests. In more detail, $\mathcal{F}_{Compute}$ accepts commands from users to create, access, or delete computing nodes. In response, it may request images from $\mathcal{F}_{Image}$, connect to volumes stored on $\mathcal{F}_{BlockStorage}$.

Compromising $\mathcal{F}_{Compute}$ gives the environment extensive power: it may create or delete arbitrary nodes from $\mathcal{F}_{Compute}$'s records and may also capture the credentials of any user who subsequently accesses the service and use these credentials to falsify requests to other services.

d) Image: $\mathcal{F}_{Image}$ stores virtual machine images that can be used when instantiating new nodes. These images may either be publicly accessible, or restricted only to users in the appropriate project. It only provides one method that $\mathcal{F}_{Compute}$ may invoke to request an image. $\mathcal{F}_{Image}$ will respond as long as credentials with appropriate permissions are provided.

Compromising $\mathcal{F}_{Image}$ allows the environment to learn both the images stored on the service as well as all user credentials that pass through it. However, a compromised $\mathcal{F}_{Image}$ cannot directly influence other services since they never expect incoming connections directly from $\mathcal{F}_{Image}$.

e) Node: $\mathcal{F}_{Node}$ is our abstraction of a virtual machine; it can execute arbitrary programs on behalf of the project that owns it. Nodes are spawned by $\mathcal{F}_{Compute}$ but then act independently. Because $\mathcal{F}_{Compute}$ sends the code of a node over the network when it is instantiated, E may view the initial code.

Compromising $\mathcal{F}_{Node}$ gives the environment the ability to view all the current executing code and to maul the computation performed within the node.

f) Storage and volumes: $\mathcal{F}_{BlockStorage}$ manages the collection of data volumes available for use by users. It provisions volumes and attaches them to nodes, but then is out of the loop during subsequent data accesses.

Compromising $\mathcal{F}_{BlockStorage}$ permits the environment to attach and detach volumes from nodes of her choice. As a countermeasure to protect the data from unauthorized disclosure, the volume can be encrypted with a key that is only known to users with the correct project permissions.

g) Message bus: OpenStack has an internal message queue to handle communication between services. It allows us to optionally enable TLS for the inter-service communication. We model the TLS-enabled message bus using a secure message transmission functionality $\mathcal{F}_{SMT}$ that protects the integrity of messages; additionally, it protects the confidentiality of tokens. This is a deviation from OpenStack as-is, which would allow any compromised service to breach message integrity and token confidentiality [36].

External communication between services and users (or their Dashboards) is handled instead by $\mathcal{F}_{ExtNet}$, which provides data confidentiality but does not authenticate the message's sender. This modeling decision reflects the fact that OpenStack never verifies whether the user sending the message is actually the owner of the credentials contained therein.

B. Security Assertions

We list below several security guarantees. We stress that this is an informal description of forbidden or 'blacklisted' activities; the UC modeling of Sections VII and VI specifies exactly the set of permissible activities in a 'whitelist' format.

A main ingredient in our modeling and analysis is the behavior of the ideal cloud upon corruption of individual services. This way, we capture the compromises we consider and the security properties we guarantee in face of compromise.

a) Authentication & authorization: As long as $\mathcal{F}_{Identity}$ is uncompromised, E is limited to perform only those actions authorized by her projects and roles. Corrupt services can perform actions within their scope on behalf of the environment, but cannot influence uncorrupted services to perform unauthorized actions.

b) User control: By moving away from bearer tokens, we can provide some user control even in the face of service-level compromises. Bearer tokens allow a corrupted service to impersonate a user to other, uncorrupted services and perform unintended actions. See Section VIII for details on our new tokening mechanism that removes the ability to replay user tokens and thus reduces the scope of a corrupted service to only those actions the service is able to perform directly. As $\mathcal{F}_{Node}$ does not have access to user tokens, a corrupted $\mathcal{F}_{Node}$ is unable to make changes affecting the OpenStack control or data plane. In this sense we model $\mathcal{F}_{Node}$ as being fully isolated from other $\mathcal{F}_{Node}$ instances or OpenStack services. Future expansion of our $\mathcal{F}_{Compute}$ model could include a hypervisor-like functionality detailing this isolation of $\mathcal{F}_{Node}$.

c) Resource control: Users may restrict the environment from accessing and tampering with computing nodes, data volumes, and images as long as two services remain uncompromised: $\mathcal{F}_{Identity}$ and the service managing the object. Put simply, the services properly separate their control and data planes. For example, a corrupted compute can delete arbitrary user nodes, but it cannot influence the data stored on unattached volumes or the actions of other nodes (e.g., request a new image from the image service) without user authorization. This guarantee holds only if E does not legitimately hold the required project/role permissions.

Note that all of the guarantees described above only apply at the OpenStack layer. For instance: if you use OpenStack to spawn a web server with several known vulnerabilities and then connect it to the Internet, it is certainly possible for E to compromise your node. We make no guarantees about the safety of objects stored *within* OpenStack, only about their management *by* OpenStack.

Additionally, enforcing these security guarantees may come at the expense of flexibility. Having all security at the border and full trust within OpenStack makes it easier to realize the cloud vision of fungibility; for instance, if one node fails then any worker can be tasked automatically to take over for it. By chaining all authorization decisions back to the user, we reduce the cloud's ability to self-regulate load balancing, scaling, and failover decisions.

C. Modeling Decisions

In this section, we discuss some of the decisions that impacted our modeling. First, we needed to decide the scope of $\mathcal{F}_{Compute}$ within Nova, the largest OpenStack service. At a high level, Nova comprises both the front-end API/scheduler and the back-end worker nodes. We choose to be more fine-grained so that our model is capable of describing the effects of compromising part, but not all of the (large) Nova codebase. This decision is made without loss of generality; compromising the entire Nova service corresponds in our mode to corrupting $\mathcal{F}_{Compute}$ and all $\mathcal{F}_{Node}$ functionalities. Second, we augment $\mathcal{F}_{Identity}$ in Section VIII to strengthen tokens so they aren't susceptible to data spills. Third, $\mathcal{F}_{SMT}$ assumes that services register keys with the message queue so that it can enforce data integrity and token confidentiality in transit on the internal network.

VI. The Ideal Cloud

Our ideal cloud functionality is a UC functionality that provides the user with the following set of commands:

a) CreateNode: Allows a user to create a new node.

b) DeleteNode: Allows a user to delete a node that they had previously created.

c) AccessNode: Allows a user to execute a command on one of their nodes.

Algorithm 1 Simplified Ideal Cloud (See the full version of this paper [22] for the detailed version)

```
1: Upon receiving (Receiver, “Delete Node”, session-id, node-id) from $E$:    ▷ Step 1
2: Send-Sim (Receiver, ”Delete Node”, session-id, user-id, node-id);    ▷ Step 2
3: Upon receiving (”Confirm”, session-id) from $S$:    ▷ Step 3
4: valid=False; NodeExist = False; Result = Fail;
5: **if** user-id is valid & user-id is allowed to delete node node-id **then**
6:     valid=True;
7: **end if**
8: Send-Sim (Receiver, ”Delete Node”, session-id, user-id, node-id, valid);    ▷ Step 4
9: Upon receiving (”Delete Node”, session-id, Continue) from $S$:    ▷ Step 5
10: **if** valid & there is node with id=node-id **then**
11:     NodeExist = True; Result = Success;
12:     Delete node-id from the Node list;
13: **end if**
14: Send-Sim (”Delete Node Completed”, session-id, valid, NodeExist);    ▷ Step 6
15: Upon receiving (”Output Delete Node”, session-id) from $S$:    ▷ Step 7
16: Output(”Delete Node”, session-id, node-id, Result) to $E$;    ▷ Step 8
```

d) AttachVolume: Allows a user to attach one of their volumes to an existing node.

e) DetachVolume: Allows a user to detach a volume that had been attached to one of their nodes.

It is, in a sense, the simplest specification that is faithful in both functionality and security to the real services. As with all UC functionalities, the simplicity of the ideal cloud is intended to promote understanding and transparency of OpenStack’s behavior. Here, we exemplify a simplified version of ideal cloud functionality $\mathcal{F}_{Cloud}$ for the Delete Node function in Algorithm 1. (The full version of this algorithm, which includes message buffering, is written in the full version of this paper [22].) Our formulation of $\mathcal{F}_{Cloud}$ is simple: $\mathcal{F}_{Cloud}$ asks the permission of the simulator S for receiving every Delete Node request, and also its permission for sending each notification back to the environment. Also, $\mathcal{F}_{Cloud}$ does not hide the user credential validity information and the node existence information, as the adversary may discover the information from the execution of requests.

The information sent from the ideal cloud to the ideal-model adversary (i.e., to the simulator) represents the information that’s allowed to be leaked. Specifically, we hide the user and service credentials from the adversary, but leak all other information. We show that, even with this advantage, the ideal cloud guarantees that the adversary cannot impersonate a user and make requests on their behalf without compromising the

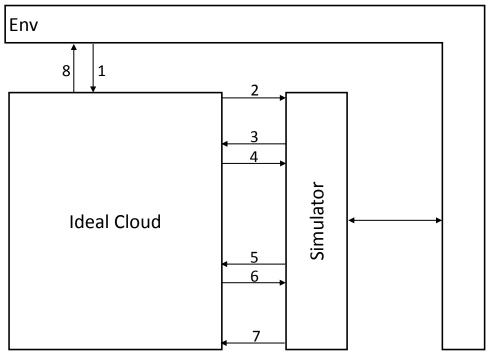

Fig. 1. Delete Node in Ideal Cloud with the Simulator.

user or services.

The ideal cloud also captures the security guarantees that are still provided in case some of the services get corrupted. In particular, the ideal cloud specifies the allowed degradation in security when a particular service is corrupted. (Notice that in the context of the ideal cloud the various services are merely names, or tags for the corruption operation made by the adversary.)

A. Ideal Cloud Walkthrough

In the ideal cloud setting, since we model the cloud as a single entity, there is no internal communication and it is left to the simulator to provide the necessary interaction with the environment. The DeleteNode request begins at Step 1 (Fig. 1) when the environment sends a Delete Node message to the ideal cloud, through a dummy user who simply forwards inputs. The ideal cloud, could simply, check that the indicated user had the correct permissions to delete the requested node and, if so, removes it from the list of active nodes. However, to capture the fact that the system leaks the user request, the ideal cloud sends the user request (without the credential) to the simulator (Step 2). When the cloud receives the confirmation message from S in Step 3 it verifies that user has permission to delete the node and relays this information to the simulator in Step 4. In Step 5, the simulator tells the cloud to continue. The cloud will then verify that the node exists and remove the node from the list of active nodes and notifies S in step 6. By receiving the continue message from S (step 7), the cloud outputs the success message to the environment through the dummy user in Step 8.

The simulator acts somewhat differently if a service (say, the Nova compute service) has been compromised. For this reason, the ideal cloud sends S a list of user-ids that have been compromised when compute is corrupted. Additionally, we observe that the environment can send any message on behalf of the corrupted compute; since the simulator cannot directly answer any requests that the environment might make to another (uncompromised) service, S must forward such requests to the ideal cloud and get a response through its specific interface. This decreases the security guarantees that are provided by the cloud.

B. Accounting for Existing Weaknesses

In the case of having compromised services, in order to UC-emulate OpenStack Services, we had to weaken the security guarantees of the ideal cloud. For example, when Nova is compromised, the adversary is able to send a request to Glance using user's credential to get an image. This means that the simulator should be able to provide the requested image to the environment. The simulator does not have the image, therefore it need to ask the ideal cloud. However, the ideal cloud does not respond to this type of simulator's requests. In order to realize the OpenStack Services, we had to remove some of ideal cloud security check points, which decreases the security guarantees. That is, for this example, when the simulator sends a request to get the image (for a corrupted user) while the user has not requested a node to be created, the ideal cloud does not check whether the user has made a "create node" request or not and instead will simply respond to the simulator. The OpenStack security imperfections discussed in the introduction under the subtitle *Security weaknesses formalized and contextualized* are the principal reasons for decreasing the security guarantees.

In order to understand why these weaknesses come into place, in the next section, we look into how real services work. We will also explore one of these weaknesses (the token mechanism) in more detail and see how an improved version yields a stronger ideal cloud (Section VIII).

VII. OpenStack Services

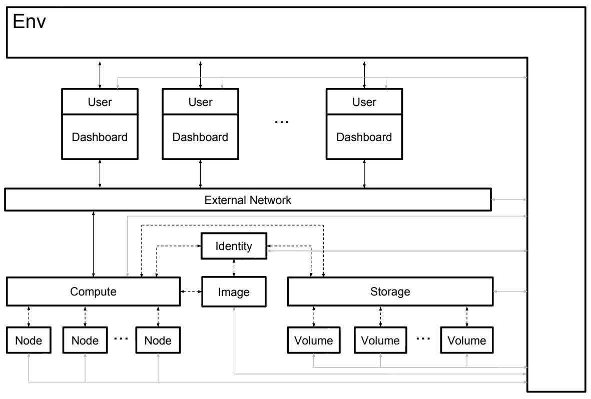

Fig. 2. OpenStack Services: Arrows indicated expected lines of communication; other communication flow is possible, but will be ignored. Gray arrows show communication with the adversary, dashed black arrows indicate communication through $\mathcal{F}_{SMT}$, and solid black arrows indicate a direct communication.

In this section we describe our model of a simplified OpenStack cloud. The service functionalities, in conjunction with the message passing functionalities, collectively provide the same set of possible commands as the ideal cloud in the previous section.

We require both that the services maintain the confidentiality of users' credentials while executing the commands and enforce that only a user in possession of the required credentials will be able to execute a command.

A. Example Workflow

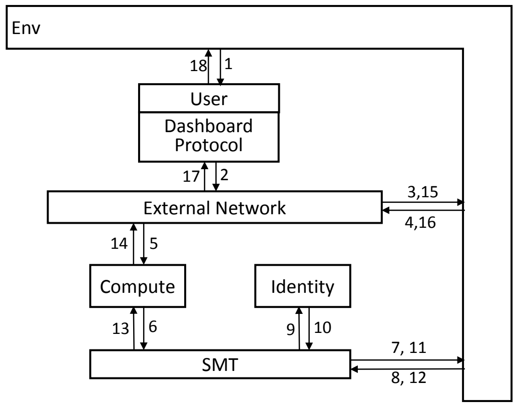

Fig. 3. The Delete node functionality starts when the environment sends a "DeleteNode" message to the user. Next the user adds its credentials and sends it to the $\mathcal{F}_{Compute}$. Then, $\mathcal{F}_{Compute}$ creates a validation request to $\mathcal{F}_{Identity}$ in order to validate user's credentials. Based on the validation result and the node existence, $\mathcal{F}_{Compute}$ deletes the node and notifies the user. We describe the workflow in detail in Section VII-A.

We will walk through an example of how a user would delete its node using these service functionalities and a dashboard protocol. This example will show the interaction between the $\mathcal{F}_{Compute}$ and $\mathcal{F}_{Identity}$ services and the user. We note that all messages between services will pass through $\mathcal{F}_{SMT}$ while any messages between the user and services will go through $\mathcal{F}_{ExtNet}$. We will briefly mention this during the description and it can be seen in Fig. 3 as well.

This interaction begins at Step 1 (Fig. 3, Algorithm 2) when the environment sends a DeleteNode message of the form ("Delete Node", session-id, node-id) to the user. The user will then reformat this message in Step 2 to include their credentials and send it to $\mathcal{F}_{ExtNet}$ to be forwarded to $\mathcal{F}_{Compute}$. When $\mathcal{F}_{ExtNet}$ (Algorithm 3) receives the message it removes the user's credentials and leaks the rest of the message to the adversary in Step 3. After receiving a (Confirm, session-id) message from the adversary in Step 4, $\mathcal{F}_{ExtNet}$ sends the original message on to $\mathcal{F}_{Compute}$. In Step 5, (Algorithm 4) $\mathcal{F}_{Compute}$ receives the message from the user and creates a request of the form ($\mathcal{F}_{Identity}$, "Service Validation", session-id, $creds_{compute}$, $creds$, "Delete node-id ") to $\mathcal{F}_{Identity}$ via $\mathcal{F}_{SMT}$ in Step 6 (Algorithm 5). The message passing by $\mathcal{F}_{SMT}$ will proceed as in Steps 3-4. After $\mathcal{F}_{Identity}$ receives the validation message from $\mathcal{F}_{Compute}$ in Step 9 (Algorithm 6) it will check that the user has permission to delete the requested node and send an appropriate response to $\mathcal{F}_{Compute}$, again via $\mathcal{F}_{SMT}$, in Step 10. Once $\mathcal{F}_{Compute}$ receives the validation message from $\mathcal{F}_{Identity}$ in Step 13, it will delete the requested node if the node exists and the validation was successful, otherwise it will note that the request failed. In Step 14, $\mathcal{F}_{Compute}$ will send a message to the user, proceeding as before through $\mathcal{F}_{ExtNet}$, notifying them of the result of their request — either the successful deletion of the node or its failure. The user, upon receiving this message in Step 17 will output the result to the environment in the final step.

B. Realization of This Workflow

We write specifications for our functionalities in pseudocode with relevant snippets for the DeleteNode workflow shown in simplified Algorithms 2, 3, 4, 5, and 6. The complete, rigorous algorithms can be find in the full version of this paper [22].

Algorithm 2 Dashboard Example Workflow

```
Upon receiving Request-Message from $E$:
User U Creates ($U_{Dashboard}$, $\mathcal{F}_{Service}$, Request-Message, creds);
Send-$\mathcal{F}_{ExtNet}$($U_{Dashboard}$, $\mathcal{F}_{Service}$, Request-Message, creds); ▷ Step 2
Upon receiving (Sender, Output-Message) from $\mathcal{F}_{ExtNet}$: ▷ Step 17
Output (Output-Message) to $E$;    ▷ Step 18
```

Algorithm 3 External Network Example Workflow

```
1: Upon receiving (Sender, Receiver, Message):    ▷ Step 2
2: Create New-Message by removing the credentials and replacing them by their IDs;
3: Send-Adversary(Sender, Receiver, New-Message); ▷ Step 3
4: Upon receiving ("Confirm", session-id) from Adversary: ▷ Step 4
5: Send-Receiver(Sender, Message);    ▷ Step 5
```

Algorithm 4 Compute Example Workflow

```
Upon receiving (Receiver, "Delete Node", session-id, node-id, creds) from $\mathcal{F}_{ExtNet}$:    ▷ Step 5
Send-$\mathcal{F}_{SMT}$ ($\mathcal{F}_{Compute}$, $\mathcal{F}_{Identity}$, "Service Validation", session-id, $creds_{compute}$, creds, "Delete node-id ");    ▷ Step 6
Upon receiving ($\mathcal{F}_{Identity}$, "Service Validated", session-id, user-id, service-id, Request, valid) from $\mathcal{F}_{SMT}$:    ▷ Step 13
Result = Fail;
if valid & a node with id=node-id exists then
    Delete node with id=node-id; Result = Successful;
end if
Send-$\mathcal{F}_{ExtNet}$ ($\mathcal{F}_{Compute}$, Receiver, "Delete Node", session-id, node-id, Result);    ▷ Step 14
```

We allow the adversary to compromise users or services (Algorithms 7 shows corrupted Compute). A compromised user or service will reveal all of its internal state to the adversary and will from that point on be under full adversarial control. In particular, compromised users and services will not necessarily follow any specified protocol and can form their own messages at will. This is particularly relevant in our case because, upon compromising a service the adversary will learn not only its credentials, which are part of the internal state of the service, but also any credentials that service learns in the future. That is, any user that makes a request to a compromised service will also leak its own credentials to the adversary.

Algorithm 5 SMT Example Workflow

```
Upon receiving (Sender, Receiver,Message):    ▷ Steps 6, 10
Create New-Message by removing the credentials and replacing them by their IDs;
Send-Adversary(Sender, Receiver, New-Message);    ▷ Steps 7,11
Upon receiving ("Confirm", session-id) from Adversary: ▷ Steps 8, 12
Send-Receiver(Sender, Message);    ▷ Steps 13, 9
```

Algorithm 6 Identity Example Workflow

```
Upon receiving (Receiver, "Service Validation", session-id, $creds_{service}$, creds, Request) from $\mathcal{F}_{SMT}$:    ▷ Step 9
if $creds_{service}$ & creds are valid & creds is allowed to perform the Request then
    valid=True; else valid=False;
end if
Send-$\mathcal{F}_{SMT}$ ($\mathcal{F}_{Identity}$, Receiver, "Service Validated", session-id, user-id, service-id, Request, valid);    ▷ Step 10
```

C. Security Analysis

The following theorem connects our UC models of the services to that of the ideal cloud.

Theorem 1. *The OpenStack Services protocol from §VII UC-realizes the Ideal Cloud $\mathcal{F}_{Cloud}$ from §VI in the ($\mathcal{F}_{ExtNet}$, $\mathcal{F}_{SMT}$)-*

Algorithm 7 Corrupted Compute

```
function ReceiveMessage
    Upon receiving (Source, message) from $\mathcal{F}_{SMT} \setminus \mathcal{F}_{ExtNet}$:
    Send-$\mathcal{F}_{SMT}$ (source, Adversary, message);
end function

function SendMessage
    Upon receiving (Adversary, "Forward", message, destination) from $\mathcal{F}_{SMT}$:
    Send-$\mathcal{F}_{SMT}$ ($\mathcal{F}_{Compute}$, destination, message);
end function

function Main
    Upon receiving a message which does not contain "Forward" & Source=Adversary from $\mathcal{F}_{SMT}$:
    Apply the request;
    Send-$\mathcal{F}_{SMT}$ ($\mathcal{F}_{Compute}$, Adversary, result of the request);
end function
```

hybrid model.

We defer to the full version of this paper [22] a rigorous specification of the all ideal services, the ideal cloud, and the simulator connecting the two. In this space, we simply highlight the main ideas involved in the proof.

Proof sketch. As with most UC analyses, our argument proceeds via induction on the steps taken by the environment E. We show that for any message that E might send, the next incoming message received by E maintains the invariant that *E's view in the ideal cloud and OpenStack Services is identical*. More specifically, for any state that E can reach, the action of the simulator S up to that point must have ensured that E's view is the same in both worlds. Ergo, E cannot distinguish whether it is interacting with the OpenStack Services or with the composition of ideal cloud and the simulator. At each step of the induction, we show that the following three properties are maintained.

1) The simulator faithfully emulates the entire workflow required to process one user-provided command, when viewed standalone.
2) The simulator faithfully emulates two interleaved user-provided commands such that the resulting state is the correct outcome after executing the two commands in an adversarially-controlled order. Put differently, there are no race conditions that cause the services to reach any kind of "weird" state.
3) An adversary who compromises services can only leverage them to damage other (uncompromised) services by leveraging OpenStack's token mechanism for authentication and authorization. Our ideal worlds accurately reflect the extent of the damage that the environment can cause by misusing acquired bearer tokens.

In the full version of this paper [22], we demonstrate these 3 properties via a tedious, yet straightforward, case analysis of all types of valid messages E can send and all actions that S is programmed to perform in response. □

Looking ahead to the next section, only property 3 of this analysis must change if the token mechanism is improved.

VIII. Improved Tokening Mechanism

Our analysis has shown that the simulator of the ideal cloud cannot emulate the corrupted services unless we relax the security promises of the ideal cloud. To a large extent, these concerns arise due to the use of bearer tokens, which enable anyone who can see a token to masquerade as corresponding user and perform any action allowed to the owner of the token on behalf of him. Bearer-tokens are leaked to the adversary in both partially and fully corrupted services which could happen through a vulnerability. Once a service corrupted, it could be malicious in at least two important ways:

- A service can use a token to obtain unrequested services charged to the user.
- A service can use a token to access other resources at another service (like exporting a disk with sensitive information).

The above problems are not tolerated by the cloud users. Having a bug free cloud is not realistic, but reducing effects of buggy code is possible. It is desired to have OpenStack guarantee the modularity security requirement defined as follows:

Definition 1. *Modularity Security Requirement*: If an adversary corrupts a service, other services just do whatever requested to do by the users, nothing more. For example, if an adversary corrupts the Image service, he cannot create or delete a node, but if a user issues a "create node" command, it is possible to create a node with a corrupted image because of the image service corruption.

We propose to limit the impact of service corruption by preventing replay attacks and scoping tokens only to handle the request desired by the user by following an idea laid out in [17], [28]: building tokens that can only be used once.

A. One Time Tokens

Within the OpenStack services, we modify all uses of tokens so that each token may only be used to perform a single request requested by the user (even if this request necessitates action by many services). Specifically, we modify Keystone to sign tokens that have been scoped by the user (and potentially scoped further down as it is passed from one service to the next) and also to maintain a blacklist of all redeemed tokens. A one-time use token scoped to a specific job resembles a "ticket" as used in Kerberos [37] or in TLS 1.3's zero round trip time mechanism [38].

Restricting tokens to be single use with specific parameters limits the capability of the adversary to misuse tokens that he captures by corrupting a service. Previously a corrupted service could store all tokens it sees and use them at will. This modification ensures that once a token has been used it will not be accepted again in the future. Additionally, since tokens are scoped, it is not possible for a corrupted service to lie to uncorrupted services about the content of a user's request. Specifically, users will indicate when requesting a token exactly which of their resources it grants access to. This would not prevent a corrupted Compute from deleting a node when a user had asked to create one or creating a node with a completely different specification since Compute could simply do these things without bothering to verify the token, but it *would* prevent Compute from requesting a different image from the Image service than the one the user had intended. This helps in the case where the adversary has not already seen, via a request by a different user, the image it would like to have Compute use. Scoped, single use tokens even benefit functionalities that go beyond those captured in our model; for example, an adversary who corrupts a VM is not able to bring down the cloud by observing user tokens in Nova message queue (this issue was discussed in [28]). We also remark that there is another notion of limited-use tokens with a security-performance tradeoff. Many OpenStack engineers desire a stateless Keystone architecture in order to distribute the authentication process over several servers for improved scalability and performance. If it is desirable to keep Keystone stateless, then one could instead design "limited-reuse" tokens with a short lifetime. While one could model the impacts of limited reuse by combining rate limiting with a UC model of network time [18], we focus on the One-Time token presented above with an explicit blacklist to prevent reuse.

B. Improved Security Analysis

We demonstrate that the improved services collectively realize a more secure cloud framework. We defer to the full version of this paper [22], where we provide a complete specification of all improved services, the new ideal cloud, and the simulator connecting the two.

Theorem 2. *The strengthened version of OpenStack Services using a one-time token securely realizes the less-leaky ideal cloud $\mathcal{F}_{Cloud\_OneTime}$ in the ($\mathcal{F}_{ExtNet}$, $\mathcal{F}_{SMT}$)-hybrid model.*

Proof sketch. The simulator in the scoped one-time token model is more stringent than the simulator in the unscoped bearer token model. To enforce the single-use constraint, S creates a black list to store the services that have already checked the validity of a credential using session-id; the simulator then rejects all future attempts to validate the same credential. To enforce the scoping constraint, S only issues credentials when given both a session-id *and* a request scope; the simulator records this intended scope and subsequently rejects all attempts to validate the credential in pursuit of a request outside of this scope.

With this augmentation to the simulator, the modularity of UC allows us to prove Theorem 2 with only a few modifications to the existing proof structure. Properties 1 and 2 from Section VII-C continue to hold in the one-time token case because those properties only demonstrate correctness when all services are uncompromised, in which case there are no tokens (of any type) for the adversary to abuse. Hence, we must only augment one property and add one more.

3') The view of E is identical in the $\mathcal{F}_{Cloud\_OneTime}$ and the OpenStack services even when services are compromised, as long as each observed token can only be used one time.
4) The cryptographic design of our stronger tokens ensures that each token may only be used once. □

C. Making Security Analyses Accessible

Juxtaposing the ideal clouds in the bearer token and one-time token scenarios can be quite illustrative. As an example, from the code of the two ideal clouds, it is easy to see that a corrupted $\mathcal{F}_{BlockStorage}$ can tamper with the delivery of an image from $\mathcal{F}_{Image}$ to $\mathcal{F}_{Compute}$ during a Create Node operation in the bearer token model, whereas the same attack is not possible in the ideal cloud for the one-time token model. Importantly, this comparison can be observed without looking at our simulators and proofs; put another way, only the ideal cloud algorithms are in the trusted code base. As a result, our analysis is accessible to the OpenStack development community, since they are already comfortable with reading code.

More generally, the UC ideal cloud that models any suggested security improvement permits OpenStack developers to understand concretely the value added by this improvement. This is an example for how UC modeling can provide developers with a valuable new tool that they can use to balance the relative importance of addressing any of the numerous bug reports on their plate at any given time.

IX. Conclusion & Future Work

This work lays the foundation for a full-scope composable security analysis for the popular cloud management framework OpenStack. It brings a number of communities together: On the one hand, our abstract model is substantially easier to absorb than the code of OpenStack and therefore opens OpenStack to a wider group within the cryptography and programming languages communities. On the other hand, our modular analysis provides the OpenStack development community with a better understanding of the security concerns surrounding some of its core design issues as well as (more importantly) a concise, tangible description of how the security of the overall cloud concretely improves by making moderate software improvements. Put differently: the modular analysis can both expose bugs and provide motivation for the developers to address them. Importantly, our approach can also be used to *assert security,* namely prove lack of flaws.

This work covers only some of the core functionality provided by a full featured cloud. In particular, there are still many remaining features necessary for fine grained access control. More work is needed in order to: (a) cover more services, (b) consider more attack (corruption) options and the security guarantees provided in these cases, and (c) analyze implementations of the various services and the associated security caveats and vulnerabilities.

While we leave these challenges to future work, we note that this last point is where the power of universal composability is put to use: our model can be extended to include internal components of services such as Compute and Storage in order to show how these subsystems combine to realize the ideal services $\mathcal{F}_{Compute}$ and $\mathcal{F}_{BlockStorage}$ that we use in this work. This level of modeling also enables clearer discussion of the effects of single-node compromises.

Furthermore, our model is easily extensible to add additional OpenStack components. There are four steps involved in adding a new service to the model: create an abstraction of its own functionality, apply local changes to other services that interact with the new service, extend the ideal cloud, and augment the simulator. Creation of a new service $\mathcal{F}_{Service}$ is modular, and the second and third steps are very simple to do. While augmenting the simulator is currently tedious and non-modular, our work shows that a domain specific language for UC functionalities could streamline this process by expressing its current programming (which is mostly case statements based upon the messages it receives) in a more abstract and event-driven style. Such a change would synergize with another direction for future research: mechanizing the analysis, especially the proofs of security.

Overall, we believe that this work provides an important benchmark to show the feasibility of modeling large-scale software packages within the framework of Universal Composability. Modularity was a crucial component toward keeping the models and analyses manageable. Indeed, this work demonstrates the value of conducting similar analyses of other software deployments in the future.

Acknowledgement

This work is supported by the National Science Foundation as part of the MACS Frontier project under NSF grants 1413996 and 1414119 (https://www.bu.edu/macs/). Ran Canetti is also supported by ISF grant 1523/14 and is a member of CPIIS.

References

[1] C. Metz, "The secret history of OpenStack, the free cloud software that's changing everything," *WIRED*, April 2012. [Online]. Available: https://www.wired.com/2012/04/openstack-3/

[2] A. Venkatraman, "OpenStack market size will cross $1.7bn by 2016, says 451 research," August 2014. [Online]. Available: http://www.computerweekly.com/news/2240226930/OpenStack-market-size-will-cross-17bn-by-2016-says-451-Research

[3] Heidi Joy Tretheway. (2017, April) Users stand up, speak out, and deliver data on OpenStack growth. [Online]. Available: https://opensource.com/article/17/4/openstack-user-survey

[4] O. G. Body, 2016. [Online]. Available: https://governance.openstack.org/

[5] B. Pfitzmann and M. Waidner, "Composition and integrity preservation of secure reactive systems," 2000, pp. 245–254.

[6] M. Backes, B. Pfitzmann, and M. Waidner, "A composable cryptographic library with nested operations," in *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, Washington, DC, USA, October 27-30, 2003*, 2003, pp. 220–230.

[7] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*. IEEE Computer Society, 2001, pp. 136–145.

[8] M. Backes, J. Dreier, S. Kremer, and R. Künnemann, "A novel approach for reasoning about liveness in cryptographic protocols and its application to fair exchange." IEEE Computer Society, 2017.

[9] F.-X. Standaert, T. G. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," in *Advances in Cryptology - EUROCRYPT 2009*, A. Joux, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 443–461.

[10] C. Kudla and K. G. Paterson, "Modular security proofs for key agreement protocols," in *Advances in Cryptology - ASIACRYPT 2005*, B. Roy, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 549–565.

[11] A. Duc, S. Dziembowski, and S. Faust, "Unifying leakage models: From probing attacks to noisy leakage." P. Q. Nguyen and E. Oswald, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 423–440.

[12] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hritcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. R. Lorch, K. Maillard, J. Pan, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Z. Béguelin, and J. K. Zinzindohoue, "Everest: Towards a verified, drop-in replacement of HTTPS," in *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, 2017, pp. 1:1–1:12.

[13] R. Canetti, S. Chari, S. Halevi, B. Pfitzmann, A. Roy, M. Steiner, and W. Venema, "Composable security analysis of OS services," in *Applied Cryptography and Network Security - 9th International Conference, ACNS 2011, Nerja, Spain, June 7-10, 2011. Proceedings*, 2011, pp. 431–448.

[14] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S. Weng, H. Zhang, and Y. Guo, "Deep specifications and certified abstraction layers," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 2015, pp. 595–608.

[15] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "Certikos: An extensible architecture for building certified concurrent OS kernels," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, 2016, pp. 653–669.

[16] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu, "Toward compositional verification of interruptible OS kernels and device drivers," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 431–447.

[17] P. Desnoyers, J. Hennessey, B. Holden, O. Krieger, L. Rudolph, and A. Young, "Using open stack for an open cloud exchange(OCX)," in *2015 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 48–53.

[18] R. Canetti, K. Hogan, A. Malhotra, and M. Varia, "A universally composable treatment of network time," in *30th IEEE Computer Security Foundations Symposium, CSF*. IEEE Computer Society, 2017, pp. 360–375. [Online]. Available: https://doi.org/10.1109/CSF.2017.38

[19] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub, "Easycrypt: A tutorial," in *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, ser. Lecture Notes in Computer Science, vol. 8604. Springer, 2013, pp. 146–166. [Online]. Available: https://doi.org/10.1007/978-3-319-10082-1_6

[20] A. Petcher and G. Morrisett, "The foundational cryptography framework," in *Principles of Security and Trust - 4th International Conference, POST*, ser. Lecture Notes in Computer Science, vol. 9036. Springer, 2015, pp. 53–72. [Online]. Available: https://doi.org/10.1007/978-3-662-46666-7_4

[21] D. A. Basin, A. Lochbihler, and S. R. Sefidgar, "Crypthol: Game-based proofs in higher-order logic," *IACR Cryptology ePrint Archive*, vol. 2017, p. 753, 2017. [Online]. Available: http://eprint.iacr.org/2017/753

[22] K. Hogan, H. Maleki, R. Rahaeimehr, R. Canetti, M. van Dijk, J. Hennessey, M. Varia, and H. Zhang, "On the universally composable security of openstack," *IACR Cryptology ePrint Archive*, vol. 2018, p. 602, 2018. [Online]. Available: https://eprint.iacr.org/2018/602

[23] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally composable security with global setup," in *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, 2007, pp. 61–85.

[24] M. Jones and D. Hardt, *RFC 6750: The OAuth 2.0 authorization framework: Bearer token usage*. Internet Engineering Task Force (IETF), 2012, https://tools.ietf.org/html/rfc6750.

[25] OpenStack Security Group, "Openstack security guide," 2015.

[26] OpenStack Foundation, "Tokens," 2017. [Online]. Available: https://docs.openstack.org/security-guide/identity/tokens.html

[27] P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace," RFC 4122 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–32, Jul. 2005. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4122.txt

[28] W. K. Sze, A. Srivastava, and R. Sekar, "Hardening openstack cloud platforms against compute node compromises," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 341–352.

[29] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono, "All your clouds are belong to us: Security analysis of cloud management interfaces," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '11. New York, NY, USA: ACM, 2011, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2046660.2046664

[30] Y. Sun, G. Petracca, T. Jaeger, H. Vijayakumar, and J. Schiffman, "Cloud armor: Protecting cloud commands from compromised cloud services," in *2015 IEEE 8th International Conference on Cloud Computing*, June 2015, pp. 253–260.

[31] Y. Sun, G. Petracca, and T. Jaeger, "Inevitable failure: The flawed trust assumption in the cloud," in *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security*. ACM, 2014, pp. 141–150.

[32] Y. Sun, G. Petracca, X. Ge, and T. Jaeger, "Pileus: Protecting user resources from vulnerable cloud services," in *Proceedings of the 32nd*

Annual Conference on Computer Security Applications. ACM, 2016, pp. 52–64.

[33] R. Canetti, S. Chari, S. Halevi, B. Pfitzmann, A. Roy, M. Steiner, and W. Venema, "Composable security analysis of OS services," in *Applied Cryptography and Network Security - 9th International Conference, ACNS 2011, Nerja, Spain, June 7-10, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6715, 2011, pp. 431–448.

[34] S. Gajek, M. Manulis, O. Pereira, A.-R. Sadeghi, and J. Schwenk, "Universally composable security analysis of tls," in *International Conference on Provable Security*. Springer, 2008, pp. 313–327.

[35] R. Canetti, D. Shahaf, and M. Vald, "Universally composable authentication and key-exchange with global PKI," in *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 9615. Springer, 2016, pp. 265–296.

[36] H. Albaroodi, S. Manickam, and P. Singh, "Critical review of openstack security: Issues and weaknesses," *Journal of Computer Science*, vol. 10, no. 1, pp. 23–33, 2014.

[37] J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in *Proceedings of the USENIX Winter Conference*. USENIX Association, 1988, pp. 191–202.

[38] H. Krawczyk and H. Wee, "The OPTLS protocol and TLS 1.3," in *IEEE European Symposium on Security and Privacy, EuroS&P*. IEEE, 2016, pp. 81–96. [Online]. Available: https://doi.org/10.1109/EuroSP.2016.18