一种新的访问控制策略描述语言及其权限划分方法

罗 杨[1)] 沈晴霓[1)] 吴中海[1),2)]

[1)](北京大学软件与微电子学院 北京 102600)

[2)](北京大学软件工程国家工程研究中心 北京 100871)

摘 要 云平台 RESTful 接口往往暴露在 Internet 上,为保证云资源的安全,防止数据泄露和非授权访问,必须实施安全策略对这些接口进行访问控制.然而,目前 RESTful 接口缺乏统一的访问控制策略描述语言及相应的权限划分机制.这导致两个问题:(1)用户不得不学习不同的策略语言来管理不同云平台上的权限;(2)云服务提供商缺乏对 RESTful 接口的细粒度的访问控制,不符合最小特权原则.对此,该文提出了一种新的访问控制策略描述语言.该语言定义了 RESTful 的标准请求格式,从而可以直接从一个 RESTful 请求中构造样本策略,为 RESTful 接口访问控制提供语法一致的策略语言.在该语言的基础上,进而提出了一种基于遗传算法的 RESTful 权限划分方法,采用 2 维矩阵表示一个权限划分,并作为遗传算法的种群个体.接着定义了选择算子、变异算子和交叉算子,提出了权限划分的三个原则:分类个数、测试用例覆盖、权限重叠,并设计适应度函数.该文基于 OpenStack 云平台给出了策略语言评估机制的参考实现,验证了方法的可行性.实验结果表明,相比 OpenStack 原有策略,该文策略评估开销降低了 19.4%.在学习成本方面,与 XACML 策略语言相比,该文策略能够减少策略管理员 41.6%的策略设计成本.该文的权限划分方法可以产生符合用户预期、可理解的划分结果,从而为云服务提供商进行权限划分提供指导.

关键词 云安全;权限管理;权限划分;权限分析;授权策略;云计算

中图法分类号 TP393 **DOI 号** 10.11897/SP.J.1016.2018.01189

A Novel Access Control Policy Specification Language and Its Permission Classification Method

LUO Yang[1)] SHEN Qing-Ni[1)] WU Zhong-Hai[1),2)]

[1)](*School of Software and Microelectronics, Peking University, Beijing* 102600)

[2)](*National Engineering Research Center for Software Engineering, Peking University, Beijing* 100871)

Abstract Cloud computing platforms usually employ representational state transfer (REST) interfaces to expose their services to the Internet, including computing service, storage service, network service, etc. To avoid data leak and unauthorized access, service providers prefer to control the access to the cloud interface through security policy enforcement. However, there is no widely-accepted standard for the authorization of the cloud public interfaces, including the security policy language and corresponding permission classification method. In a cloud, besides the cloud provider, the tenants can modify his own policy too. Without a unified authorization language, the tenants have to learn and design different security policies if they want to use multiple clouds. To address this issue, in this paper, we propose a novel access control language to control the access to a cloud interface. An automatic policy generation algorithm is proposed to

收稿日期:2017-05-29;在线出版日期:2017-10-24.本课题得到国家自然科学基金(61232005,61672062)、国家"八六三"高技术研究发展计划项目基金(2015AA016009)资助.**罗 杨**,男,1989 年生,博士研究生,中国计算机学会(CCF)会员,主要研究方向为大数据与云计算安全.E-mail:luoyang@pku.edu.cn.**沈晴霓**,女,1970 年生,博士,教授,博士生导师,中国计算机学会(CCF)会员,主要研究领域为操作系统安全、大数据与云计算安全.**吴中海**(通信作者),男,1968 年生,博士,教授,博士生导师,中国计算机学会(CCF)高级会员,主要研究领域为大数据与云计算安全、分布式机器学习系统.E-mail:wuzh@pku.edu.cn.

《计算机学报》

automatically generate access control policies from the cloud requests. It reduces the human intervention in the policy design process. The generated policy can be used to assign permissions to certain groups or roles for fine-grained access control. So that when one administrator account is compromised, the adversary can only utilize the permissions that are assigned to that administrator, which reduces the attack surface. A permission classification method based on matrix operations is proposed to solve the permission classification issue. The integration test is an important input of our algorithm. The dependency on the integration test will not affect our method's applicability. Because nowadays, most of the large-scale software like a cloud platform is already shipped with a complete integration test set. Our permission classification method supports three goals: expected number of classification groups, the coverage of the original management tasks and the number of permission overuse. The expected number of classification groups can be customized by the cloud provider. The coverage of original management tasks can reflect how complete the resulting categories of permissions can cover the original cloud management tasks. The number of permission overuse reflects how the resulting categories conform to the principle of least privilege. These three goals contradict with each other and it is difficult to propose a straightforward algorithm to get a balanced permission classification result for the three goals. So we choose to use a heuristic algorithm. We implement a prototype of our policy language's enforcement mechanism on a popular open-source cloud platform called OpenStack to show the effectiveness and performance of our method. Compared to the original OpenStack policy, the enforcement overhead of our policy is reduced by 19.4%. In the usability, we compared our language to a popular authorization policy language called XACML. By investigating an experiment on policy designers from different levels, we find that our language can cut down the learning cost for about 41.6% compared to XACML. Our permission classification method can also provide a reasonable classification result for permissions. They can be used as an important reference when the cloud provider manages the security tasks for a cloud. The overhead of running the algorithm is acceptable in a real cloud environment.

Keywords cloud security; permission management; permission classification; permission analysis; authorization policy; cloud computing

1 引 言

目前,REST(REpresentational State Transfer)[1]已经成为 Web 服务接口的事实标准.符合 REST 标准的系统称之为 RESTful.近年来,随着虚拟化等技术的发展,云计算已经成为企业减少运营成本,实现按需管理计算设施的重要途径[2].目前的主流云平台都通过 RESTful 接口提供云服务.由于这些接口通常直接暴露在 Internet 上,因此,为了防止数据泄露和非授权访问,有必要对这些 RESTful 资源进行访问控制.

云平台 RESTful 接口的访问控制,主要是通过访问控制策略的形式来实施.近年来,学术界提出了多种访问控制策略描述语言,如 XACML(eXtensible Access Control Markup Language)[3]、SPL(Security Policy Language)[4]、Ponder[5]等.例如目前较为主流的 XACML 语言,由标准组织 OASIS 提出,采用 ABAC(Attribute-Based Access Control)模型[6].然而,除了 JBoss,Axiomaics,OpenAZ 等系统外,XACML 并没有被业界云平台广泛应用.大型公有云提供商如 Amazon Web Services(AWS)、Microsoft Azure 等更倾向于自行设计一套策略描述语言用于自身平台.类似还有 HP Openview PolicyXpert、CiscoAssure 等网络管理平台,也都仅支持自己的策略语言[7].虽然目前多策略、混合策略成为访问控制策略领域的研究热点[8-10],然而现有解决方案仍然侧重于提高策略描述语言的表达能力和功能.大多数存在语法复杂、实现复杂度过高、难以直接适配实际环境等问题.

在利用策略描述语言表达好权限以后，接下来的一个问题就是如何对权限进行划分，细化访问控制的粒度.当前部分云平台的权限管理在控制粒度方面较差，如 OpenStack 作为业界主流的开源云平台，默认只有两个角色：管理员角色和普通用户角色.云服务提供商通常采用内置的管理员角色对云平台进行配置和管理.这样的设计造成单一管理员拥有云平台的所有权限，能够无限制访问任意云中数据，如虚拟机、网络、磁盘、镜像等.一旦管理员账户被外界攻击者攻破，或者出现内部攻击者，则云平台整体的安全性都失去了保障.

为了缓解这一攻击，可以采用细粒度的访问控制模型.以 OpenStack 为例，OpenStack 采用 ABAC 和 RBAC（Role-Based Access Control）模型[11-12]的一个子集进行访问控制.云服务提供商可以设计 ABAC 或者 RBAC 授权规则来控制对云平台 RESTful API 的访问.云平台 RESTful API 中有一部分是由云平台自身的管理人员调用，用来维持云平台运行的操作，我们称之为管理 API 或特权.如何将云平台管理 API 的访问权限划分成不同的子集，分配给多个 RBAC 角色，是一个难点.因为这需要专业的安全知识，并对云平台的所有操作都有一定了解.另外，对成百上千的 RESTful API 进行人工划分也是一个很耗时的操作.

目前在 RBAC 研究领域，通常借助角色工程来解决角色分配问题.角色工程是一个定义 RBAC 角色、权限、约束和角色层次的过程[13].大多数的角色工程方法在角色和权限之间插入了中间层，如场景、工作流等，来帮助实现从权限到角色的映射.然而，这些方法通常需要大量的人工参与.并且还存在一个适用性的问题.比如，如果一个系统只支持 RBAC0 模型[12]（如 OpenStack），如果角色工程的结果包含了 RBAC 的高级特性，比如角色层次，由于角色层次是在 RBAC1[12]中才加入的高级特性，因此会导致该角色工程结果无法在 OpenStack 这种低级别 RBAC 模型中应用.因此有必要讨论 RESTful 场景下的角色工程问题，即如何将现有的 RESTful 接口函数划分为若干类，既满足最小特权原则[14]，也能满足安全管理员的个性化需求（如期望的分类个数）.

为了解决 RESTful 接口中存在的访问控制问题，本文提出一种面向 RESTful 接口的访问控制策略描述语言 RestPL（REST Policy Language），定义了 RestPL 语言的标准请求格式、语法元素以及策略自动生成算法，降低了策略设计的开销.同时，为了减少 RestPL 策略中权限划分方面的人工参与，本文提出一种基于遗传算法的自动化 RESTful 权限划分方法.该方法将云平台集成测试集定义为测试矩阵，将权限分类结果定义为分类矩阵.然后定义遗传算法的几个要素，包括个体、交叉算子、变异算子、选择算子、适应度函数等，给出遗传算法的终止条件.本文提出的 RestPL 策略语言及其权限划分方法在云平台 OpenStack 上进行了实现.首先，本文在 OpenStack 上测试了 RestPL 的策略评估性能.实验结果表明，与 OpenStack 原有策略相比，RestPL 的策略评估开销降低了 19.4%.与 XACML 相比，RestPL 具有更低的学习曲线，平均的策略编写成本降低了 41.6%.在权限划分方面，在 OpenStack 上的实验结果表明，本文方法能够获得 RESTful 权限划分的近似最优解，并且该权限划分结果是人可理解的.本文同时也对权限划分的性能进行了评估，测试阶段的开销为 13.1%，遗传算法阶段的开销为 183.29 s.由于权限划分不是一个经常性的操作，因此该开销是可以接受的.

2　相关工作

近年来，学术界提出了很多访问控制模型和策略描述语言.基于属性的访问控制（ABAC）[15]将基于角色的访问控制（RBAC）[12]中的角色扩展为更为通用的属性概念，增强了模型的表达能力.标准组织 OASIS 提出了一种声明式的访问控制策略语言 XACML[3].XACML 基于 ABAC 模型，定义了策略集、策略、规则的概念，并设计了拒绝优先、允许优先等组合算法来实现规则间、策略间的冲突消解.XACML 同时还提出了包括策略执行点（Policy Enforcement Point，PEP）、策略决策点（Policy Decision Point，PDP）、策略信息点（Policy Information Point，PIP）等模块的参考架构设计.XACML 的策略、请求和响应都采用 XML 格式进行描述.然而 RESTful 接口更倾向于采用简洁的 JSON 语法，两者之间存在一定差异.

文献[4]提出的 SPL 语言，支持实体、关系、属性、量词的比较.文献[16]提出的基于策略的访问控制（Policy-Based Access Control，PBAC）能够控制移动应用的权限.

文献[17]提出了基于道义逻辑的策略描述语言 Rei，以满足动态、开放环境中的安全需求.在 Rei 中，

策略以约束的形式规定可在资源上进行的授权或义务性操作. Rei 通过元策略解决策略一致性和冲突问题. 该语言支持组、角色等表达特性.

文献[18]提出了企业级隐私授权语言 EPAL (Enterprise Privacy Authorization Language). EPAL 更侧重于隐私保护而不是访问控制,应用没有 XACML 广泛. EPAL 采用基于目的的访问控制模型(Purpose-Based Access Control). 例如,当医生具有治疗诊断目的时,就可以访问对应病人的信息. 该模型简化了传统的 RBAC 模型,但引入了额外的目的元素. 该语言提供了一定的策略精化和冲突检测机制.

文献[19]提出了基于一阶逻辑的授权描述语言 ASL(Authorization Specification Language). 该语言具有用户、角色、客体等 RBAC 模型中的概念. 授权规则是从{用户,角色集,客体,动作}四元组到决策结果(允许,禁止)的映射. ASL 策略需要采用专门的逻辑语言 Datalog 来编写,其策略评估和冲突检测可以通过逻辑推理保证完备性. 在性能方面,ASL 的策略评估开销与规则数目成正比.

在 RBAC 角色工程领域目前也有很多研究. 文献[20]提出了一种基于流程的角色工程方法. 文献[21]提出了一种基于 UML(Unified Modeling Language)的角色工程方法,可以将 RBAC 模型用 XML 图表来描述. 文献[22]则提出了一个三层模型来实现角色到权限的映射,三层分别为任务、工作模板和工作. 然而这样的三层概念对初学者来说理解起来比较困难,难以直接在实践中应用. 文献[23]提出了一个基于用例的角色工程方法. 用例需要采用 UML 语言描述,并满足特定安全约束. 授权规则则可以从这些用例中生成. 文献[24]提出了一个基于场景的角色工程方法. 为了实现某个场景,访问控制的主体必须拥有执行场景中每个步骤的权限. 这个方法与本文方法的相似之处在于,都采用了用例或者测试用例来指导权限划分. 然而,该文的用例需要人工去编写,而本文所依赖的测试用例是现代软件工程中的一个必备的组成部分,因此不需要额外的人工开销. 此外,本文采用的遗传算法可以平衡多个权限划分原则,并给出一个折中的划分方案,而不是仅仅考虑用例.

3 云服务案例分析

目前,RESTful 接口在多个应用场景下都面临着访问控制的问题,如云计算[25]、在线办公管理[26]、在线医疗等. 由于篇幅限制,本文仅以云计算场景为例来阐述 RESTful 接口在访问控制策略描述语言方面所存在的问题.

3.1 云服务场景

下面给出一个配置安全策略来获取云平台虚拟机资源的实例:假设一个企业采用混合云的方式来部署业务,公有云部分采用目前业界规模最大的公有云 AWS,私有云部分则采用目前主流的开源云平台 OpenStack 搭建. 该企业所在的租户是 TENANT1,需要授权一个名为 USER1 的员工同时对 AWS 中的虚拟机 VM1 和 OpenStack 的虚拟机 VM1 进行获取基本信息这样一项管理操作,因此可得到以下高层业务需求:

TENANT1 中的 USER1 想要获取当前租户下的虚拟机实例 VM1 的详细信息.

上述需求需要由企业的安全管理员制定具体的访问控制策略来进行实施.

3.2 现有策略的不足

访问控制需求的目的是允许某一类 RESTful 访问请求的访问,因此高层业务需求与具体的 RESTful 访问请求是相对应的. 针对上节提出的需求,在 AWS EC2(Elastic Compute Cloud)云平台中实际的访问请求如下:

```
GET
https://ec2.amazonaws.com/?Action=DescribeInstances&
Filter.1.Name=instance-id&Filter.1.Value.1=VM1&
AUTHPARAMS
```

AWS 云平台提供了 IAM(Identity and Access Management)访问控制组件进行云资源的权限管理. IAM 允许租户采用 AWS 定义的一套策略描述语言定义自身的访问控制策略. 为了使上述访问请求能够被 IAM 允许通过,该企业的管理员需要在 AWS IAM 中设计一条策略,对 VM1 的查询操作进行授权,并且需要把该策略绑定到主体 USER1 上. 具体的策略如下:

```
{
  Version: 2012-10-17,
  Statement: [
  {
    Action: ec2: DescribeInstances,
    Effect: Allow,
    Resource: arn:aws:ec2:::VM1
  }
  ]
}
```

假设 OpenStack 私有云的域名为 www.test-cloud.net，由于 OpenStack 遵循 REST 设计规范，针对上节提出的需求，在 OpenStack 云平台中实际的访问请求如下：

```
GET
http://www.test-cloud.net/v2/TENANT1/servers/VM1
```

OpenStack 的策略配置文件 policy.json 本身也提供了一套类似策略描述语言的语法. 为了使上述请求在 OpenStack 中被权限检查组件允许通过，需要在上述策略配置文件中定义如下策略：

```
{
 compute:get_all: project_name:TENANT1 and user_name:
 USER1 and TENANT1:%(project_name)s and VM1:
 %(instance_name)s
}
```

在上述策略中，compute:get_all 表示获取虚拟机信息这一动作，project_name 是主体所属的租户，user_name 是主体标识符. %(project_name)s 是客体所属的租户，%(instance_name)s 是客体标识符.

从上述比较中可以发现，OpenStack 和 AWS 都提供了各自的访问控制策略描述语言. 由于其设计不同，导致两者在语法和语义上存在不小差异，也给用户的学习、理解造成困难. 如 AWS IAM 策略是绑定到主体上的，策略规则内并没有显式指定主体，只有 Action，Resource，Effect 等属性，Effect 可以为 Allow 或者 Deny. 而 OpenStack 的策略则显式指定了访问控制主体，每一条策略规则是键值对的形式，键匹配动作，值为匹配的条件，条件中可以指定主体属性或客体属性. OpenStack 策略中没有 Effect 这个元素，可以认为其值永远为 Allow，即匹配规则即允许，没有匹配则禁止.

可见，由于混合云中的两个云平台所采用的访问控制策略描述语言并不相同，因此为了描述同一需求，该企业的策略管理员需要学习和使用两种截然不同的策略描述语言，这无疑增加了人工成本. 另外，除了云计算场景外，该企业可能对其他类型的 RESTful 服务也有需求，策略管理员不得不学习该服务所指定的策略语言，来实现访问控制目标.

4　权限划分问题背景

4.1　REST 标准

REST 由 Fielding 在 2003 年在其博士论文中提出[1]，目前已成为广泛认可的 Web 服务接口标准. 符合 REST 标准的系统通过 URI 向外界提供 Web 资源，并采用 HTTP 协议标准的方法作为操作. 例如，如果要删除一个叫/people/tom 的资源，则可以采用 HTTP 标准的 DELETE 方法请求/people/tom 地址. 这种符合 REST 规范的 Web 服务 API 称为 RESTful API. RESTful API 的基本格式可以定义如下：

(1) 资源路径，如出现在网址 http://example.com/books/book1 中的/books/book1；

(2) 动作，通常是 HTTP 协议标准中的 Method 字段，包括 GET，PUT，POST，DELETE 等，与常见的查、改、增、删等操作相对应.

一个抽象资源路径是指不包含个体资源信息的资源路径. 例如，DELETE /users/〈USER〉是一个用来删除用户的 RESTful 函数，〈USER〉用来指代一个用户，比如 tom. 本文则将 DELETE /users/〈USER〉称为一个抽象资源路径. 抽象资源路径与动作在一起构成了一个 RESTful 函数. 本文提出的权限划分方法就是以 RESTful 函数为划分对象的.

4.2　权限划分原则

为了遵循最小特权安全原则，有必要将代表管理权限的 RESTful 管理 API 进行划分，分成几个权限子集，RBAC 角色分别授予不同的管理员. 这样，每个管理人员通过对应的角色只掌握部分权限. 当攻击者攻破一个管理员账户时，也只能拿到云平台的一部分权限，造成有限的破坏. 因此，权限划分技术缓解了针对云平台管理平面的攻击，有效限制了攻击范围，为云平台争取了更多的时间察觉该攻击并采取防御措施. 在本文中，权限划分的结果就是权限类的集合，权限类则是 RESTful 函数(或者叫权限)的集合，权限类之间的权限可以重叠. 所有权限类的并集应该等于所有需要被划分的权限，即每个权限至少被划分到一个类中.

本文提出了权限划分的几个原则：

原则 1.　划分出来的权限类的个数应该保持在合理范围内. 例如，如果云服务提供商期望安排 5 个云平台管理员，权限划分方法则需要将所有的管理 API 尽可能地划分成 5 部分.

原则 2.　权限划分不应该影响原有云平台管理任务的执行，如虚拟机迁移、IP 地址分配等. 如果将每一个管理任务看成一系列 RESTful 函数的调用，那么权限划分后，针对每一个管理任务，至少应该有一个管理员能够有权限执行该管理任务中所有

《计算机学报》

RESTful 函数. 这样,每一个管理任务都会有管理员能够去独立完成,不会出现需要两个管理员相互配合完成同一个任务的情况,从而避免影响原有任务的执行.

原则 3. 不同权限类之间的权限重叠应尽可能少. 如果权限划分后,两个管理员共享很多权限,说明他们的职责是类似的,并没有区分开,这样不符合职责分离原则. 同时将同一个权限分配给了多个管理员,也增加了攻击面. 攻击者只要攻破多个管理员账号中的一个,则可以拥有该权限.

总体来说,权限划分存在两个难点:(1) 如何确定需要保证运行的管理任务;(2) 如何进行权限划分. 对于问题(1),本文提出了一个基于集成测试的方法,将在后文中进行论述. 对于问题(2),本文发现很难给出一个方案对权限直接进行划分,但是当给定一个划分方案时,却比较容易依据上述的三个原则评价该方案的好坏. 后者其实是一个 NP 问题:假设目前有 n 个权限需要划分成 k 类,每个类中至少包含一个权限,任意两个类之间权限不可重复. 则权限分类的情况的个数是第二类 Stirling 数[27]. 第二类 Stirling 数就表达了将 n 个物体划分为 k 个非空子集的方法个数:

$$S(n,k)=\frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n \tag{1}$$

可以看出分类的时间复杂度是 n 的指数级别:$O(k^n)$. 如果类之间的权限允许重复的话,结果则会更多. 因此,穷举所有的分类找到最优解在性能上是无法接受的. 因此需要找出一种优化方法在有限时间内找到一个权限分类的近似最优解. 因此,本文提出了一种基于遗传算法的权限划分方法,能够显著降低解空间搜寻的时间开销.

4.3 遗传算法

遗传算法是一种模拟了自然选择进程的启发式算法,属于进化算法大类中的一种. 遗传算法通过依赖类似生物学上的变异、交叉、选择等操作搜索问题空间,并得到高质量的解[28].

遗传算法中的进化通常从随机产生的初始种群开始进行迭代. 每一次迭代中,种群会经过变异、交叉等操作生成新的个体,类似于生物学上的基因变异、遗传等现象. 然后基于预先给定的适应度函数计算当前种群每个个体的适应度. 适应度函数可以对个体进行评价打分,符合要求的个体分数会更高. 然后通过选择操作淘汰一部分适应度差的个体,选择适应度较高的个体形成新种群,进入到下一轮迭代. 迭代的停止条件通常为到达迭代次数上限或者找到满足目标适应度的个体.

4.4 集成测试

本文提出的权限划分方法需要依赖集成测试作为先验知识. 目前主流的软件测试流程主要包含单元测试和集成测试. 单元测试单独对每一个软件模块进行测试,而集成测试则对各个软件模块组成的一个整体进行测试. 通过在集成测试中编写测试用例,可以发现软件整体在接口和功能上的错误. 以 OpenStack 为例,Tempest 是 OpenStack 的集成测试集,用来测试 OpenStack RESTful API 的功能. Tempest 力求完全覆盖 OpenStack 的 RESTful API,并且包含常用的使用场景.

集成测试中的每个测试用例都会测试一部分 RESTful API,来实现对不同功能的测试. 因此可以认为,测试用例将其所测试的 RESTful API 函数的集合映射到每一个管理任务. 管理员能以自身权限独立完成被赋予的管理任务,也就意味着管理员的权限应该包括测试用例中所包含的全部 RESTful 函数,即总有一个权限类包含了测试用例所包含的权限. 这时,我们称该权限划分覆盖了测试用例.

5 RestPL 语言设计

本文提出的 RestPL 应该具备以下 3 个特点:

(1) 通用的:RestPL 应该天然适用于 RESTful 接口所需要的访问控制任务. RestPL 设计时保证了其在应用于任意 RESTful 接口时,都具有相同的语法和相似的语义. 甚至 RESTful 接口的提供商可以直接在他们的系统中复用 RestPL 所提供的参考实现以及相关工具(如策略编辑器等). 由于 RestPL 是专门针对 RESTful 接口设计的策略描述语言,因此并不适用于其他接口,如 SOAP 等. 由于目前 REST 已经成为目前 Web 接口的事实标准,因此 RestPL 的这一局限并不会造成很大影响;

(2) 面向请求的:RESTful 服务的客户应该可以只通过 RestPL 这一种语言对其所使用的所有 RESTful 接口进行权限管理工作. 原先复杂的策略设计流程得到简化,用户甚至可以通过构造一个 RESTful 请求直接生成一个样本 RestPL 策略,然后再经过少量的修改就可以将该策略投入正式使用;

(3) 语法简单的:策略语法应支持现代的 JSON

格式以及 RBAC 角色，从而降低用户学习成本.

下文将首先定义标准的 RESTful 请求格式，然后围绕标准请求格式，定义 RestPL 语言的语法元素.

5.1　标准请求格式

REST 是一种适用于 Web 接口的抽象的设计模式，其并不规定 RESTful 请求的具体语法和语义. 这导致不同服务提供商的 RESTful 请求格式并不相同. 基于上文对主流云平台接口的分析，可以提取出典型 RESTful 请求的关键特征，继而提出标准请求格式(Standard Request Form，SRF)，SRF 的目的是实现对绝大部分现有 RESTful 请求的兼容. 由于 RestPL 是一个面向请求的策略描述语言，因此可以直接在符合 SRF 请求规范的 RESTful 应用中使用. 这里给出 SRF 的定义：

Verb

{http|https}://Domain[/Version]/Object/?Query

Verb：HTTP 协议方法，包括 GET，POST，PUT，DELETE 等；

Domain：表示 RESTful 服务所在的域名. 目前一部分服务提供商采用域名前缀来标识不同的服务. 如公有云 AWS 采用 ec2.amazonaws.com 标识计算服务 EC2，而用 s3.amazonaws.com 标识对象存储服务 S3. 然而，也有部分 Web 服务采用网络端口来标识服务. 如开源云平台 OpenStack 采用 8774 端口来标识计算服务 Nova，6000 端口来标识存储服务 Swift. RestPL 的 Domain 字段也可以在域名后包含端口号，因此 RestPL 语言对于域名前缀、端口号两种服务指定方式都支持.

Version：表示该 RESTful 接口的版本. 在 RESTful 调用中提供版本对于同时维护多个版本的 RESTful 接口是有益的，调用者可以通过指定 Version 来确定所调用接口的版本. 另外，由于版本不同可能会造成具体的请求格式也不同，其对应 RestPL 策略也会不同，因此在 RestPL 中指定 Version 则可以针对不同版本的 RESTful 接口进行权限控制.

Object：表示请求中的 URI 路径. 由“/”分隔，可以代表一个或一系列 Web 资源的 ID 或名称.

Query：表示请求的参数或条件. 由若干个键值对组成，具体语法为

?Query_Key1=Query_Value1&Query_Key2=Query_Value2&..

需要注意的是，目前 SRF 中还没有定义访问控制的主体 Subject. 这是由于访问控制主体的具体形式通常与身份认证机制密切相关，如 HTTP 基本认证(HTTP Basic Authentication)将主体存储在 HTTP 请求头中的 Authorization 字段中，而 OpenStack 则采用其身份认证组件 Keystone 颁发的访问令牌作为访问者的标识. 因此，Subject 需要结合具体的系统进行获取，下文会阐述该问题.

5.2　语法元素

由于 RestPL 面向请求的特征，其策略与上述定义的 SRF 之间有很多相似之处. RestPL 策略在顶层由两个基本元素构成：Version 和 Statements. Version 已经与上文 SRF 中定义的保持一致. Version 字段是可选的，当一个具体请求不包含 Version 时，策略中也不必指定 Version 字段. 策略中另一个重要元素是 Statements 字段. Statements 由若干的 Statement 构成. Statement 则用来描述授权规则，即主体 Subject 能够在 Query 指定的条件下以动作 Verb 访问资源 Object，Statement 具体定义如下：

[Subject,] Object, Verb, Query→Effect

在上述定义中，Effect 表示 Statement 规则所对应的决策效果. 由于 RestPL 同时支持正向和负向授权. Effect 可以指定为 Allow 或者 Deny. 综上，RestPL 的语法规则可以表示成如下 BNF(Backus-Naur Form)范式：

Policy::=[Version,] Statements

Statements::={Statement}

Statement::=[Subject,] Object, Verb, Query→Effect

Subject::=Domain:(User|Role)

Object::=/path1/path2/../path*n*

Verb::=GET|POST|PUT|DELETE

Query::={Query_Item}

Query_Item::=Query_Key:Query_Value

Effect::=Allow|Deny

由于 Subject 不是 SRF 的一部分，因此 RestPL 策略也就无法直接在 SRF 中取得 Subject. RestPL 语言将 Subject 字段设为可选字段，并按照策略中是否指定 Subject 分两种情形讨论：(1) 策略中指定 Subject：如果服务提供商在 RestPL 策略中设置了 Subject 字段，则需要符合 Domain:User|Role 的格式. Domain 是租户的标识符，通常代表用户或角色所在的组织. User 是用户的标识符，用来指定具体某个用户所实施的策略，而 Role 是角色的标识符，用来指定是针对哪一个 RBAC 角色实施该策略，在 RestPL 策略中，User 与 Role 只须指定其中之一. 可以看出，RestPL 语言通过 Role 实现了对 RBAC 模型的支持；(2) 策略中不指定 Subject：如果服务提

供商不希望在 RestPL 策略中显式指定主体，则可以采用策略绑定主体的形式，即每一个策略规则需要绑定到具体某个主体时才会生效，同时该策略也隐式指定了被绑定的主体. 本文接下来主要以策略中指定了 Subject 的情况进行阐述.

5.3 策略自动生成算法

由于 RestPL 语言面向请求的特性，可以通过构造一个 RESTful 请求自动地生成一个 RestPL 样本策略，该策略的 Effect 为 Allow，只对该特定 RESTful 请求实现允许授权. 以该策略为基础，通过修改部分字段就可实现自定义的访问控制策略. 策略自动生成的好处是不必从头设计策略，从而减轻了安全管理员编写策略的负担. 本文提出的 RestPL 策略自动生成算法如下：

算法 1. 策略自动生成(Automatic Policy Generation).

输入：RESTful 访问请求 r

输出：RestPL 策略 p

1. $verb, domain, object, query = parseRequest(r)$;
2. $p = newPolicy()$;
3. IF ($version$ exists in r) THEN
4. p.Version = $parseVersion(r)$;
5. ENDIF
6. $s = newStatement()$;
7. IF ($user$ exists in r) THEN
8. $user = parseUser(r)$;
9. s.Subject = $domain:user$;
10. ENDIF
11. s.Object = $object$;
12. s.Verb = $verb$;
13. s.Query = $query$;
14. s.Effect = Allow;
15. $p.addStatement(s)$;

5.4 云服务需求描述

为了验证 RestPL 的表达能力，本节将第 3 节提出的云服务需求采用 RestPL 进行描述. 根据 5.3 节的策略自动生成算法，通过解析云服务案例中的 AWS 请求和 OpenStack 请求，可以自动化地得到如下 RestPL 策略：

(1) 与 AWS 请求相对应的 RestPL 策略：

```
{
  Version: 2012-10-17,
  Statements: [
  Subject: {
    Domain: TENANT1
    User: USER1
  },
  Verb: GET,
  Queries: {
    Action: DescribeInstances,
    Filter.1.Name: instance-id,
    Filter.1.Value.1: VM1
  },
  Effect: Allow
  ]
}
```

(2) 与 OpenStack 请求相对应的 RestPL 策略：

```
{
  Version: v2,
  Statements: [
    Subject: {
      Domain: TENANT1,
      User: USER1
    },
    Object: TENANT1/servers/VM1,
    Verb: GET,
    Effect: Allow
  ]
}
```

由于 REST 本身只是风格规范，不同 RESTful 接口在实现时可能采用不同的表达方式，如客体既可以在 Query 字段中指定，也可以在 Object 字段中指定，从而导致上述两条 RestPL 策略在表达上并不完全相同，但可以看出其仍具有相同的语法和语义.

综上，RESTful 接口采用 RestPL 描述访问控制策略具有两个优势：(1) RESTful 服务提供商不再需要设计一套访问控制策略描述语言以及相对应的决策机制，可以采用 RestPL 及其评估机制的参考实现；(2) 用户可以只通过一种策略语言实现对包括云服务在内的多种 RESTful 服务的权限管理，甚至可以采用相同的工具集，如策略编辑器等模块来实现对不同服务提供商的 RestPL 策略的图形化编辑等功能.

6 权限划分方法

根据 4.1 节可知，RESTful 权限由 RestPL 语言的 Verb 和 Object 字段两部分组成. 可以通过指定 RestPL 策略的 Role、Verb 和 Object 等字段实现对权限组的授权，即将全部权限划分为若干组，然后将每一组权限对应分配给一个角色. 然而，如何实现

《计算机学报》

RESTful 权限的划分则是一个需要解决的问题. 本节提出一种基于遗传算法的自动化权限划分方法，可以将 RESTful 权限映射为角色，从而方便 RestPL 策略的制定.

6.1 概　述

由于 4.2 节提出的权限划分三个原则是相互矛盾的，难以找到一种直接的方法或算法得到最优的权限划分. 例如根据原则 2，权限划分应支持原有管理任务，则权限划分的结果会倾向于分成较多个权限类，每个权限类倾向于具有较多的权限；然而根据原则 3，权限类之间应减少重叠，在本文中体现为重叠计数最少，权限类之间每出现一个重叠权限则重叠计数加 1. 因此权限划分的结果会倾向于分成较少个权限类，每个权限类倾向于具有较少的权限. 这样原则 2 与原则 3 之间就是完全对立的. 而原则 1 要求权限类个数接近用户输入的期望分类个数，则对权限划分得到的权限类个数直接进行了限定，个数过多或者过少都会影响原则的实施. 因此原则 1 与原则 2，3 也有对立之处. 针对单独一个原则比较容易提出一个直接的权限划分方案，但是如果要提出一个直接的方案来实现三个原则的权衡，则比较困难. 因此一个较好的办法就是，将分类结果定义为个体，综合这三个原则设计一个目标函数，然后利用遗传算法搜索出目标函数取值最高的分类. 虽然在有限时间内，得到的结果可能不是解空间里的最优解，但通常都是一个近似最优解. 本节首先提出分类矩阵和测试矩阵的概念，并将分类矩阵定义为遗传算法的个体，接着给出遗传算法的初始种群、各个算子、适应度函数等.

6.2 分类矩阵

本文用二维矩阵来描述对 RESTful 函数的分类结果，称为分类矩阵(Classification Matrix，CM). 矩阵的每一行代表一个分类，每一列则代表一个 RESTful 函数，元素取布尔值 0 或 1. 1 代表当前行所代表的分类包含当前列所代表的 RESTful 函数，0 则代表不包含.

下面给出一个简化后大小为 4×12 的 OpenStack Nova 的 CM 作为例子：

$$CM=\begin{pmatrix}1&1&1&0&0&1&1&0&0&0&1&1\\0&0&0&1&1&0&0&0&0&0&0&1\\0&0&0&0&0&0&0&1&0&0&0&1\\0&0&0&0&0&0&0&0&1&1&0&0\end{pmatrix}.$$

上面的 CM 代表将 12 个 RESTful 函数分成了 4 类. 类之间没有区别，因此 CM 的各行之间互换后还是等价的. 上述 CM 的 12 个列定义如下：

1. flavors | GET
2. flavors | POST
3. flavors/%NAME% | DELETE
4. limits | GET
5. os-agents | GET
6. os-agents | POST
7. os-aggregates | GET
8. os-aggregates | POST
9. os-floating-ips-bulk | GET
10. os-hosts | GET
11. servers | GET
12. servers | POST

其中，第 1 类包含包括函数 1，2，3，6，7，12. 第 2 类包括函数 4，5，12. 第 3 类包括函数 8，12. 第 4 类包括函数 9，10. 接下来我们讨论 CM 应满足的两个条件：

CM 规则 1：行数上限. 一个 CM 矩阵的行数是有上限的. 由于 CM 代表如何将列中的 RESTful 函数分成用行表示的类，分类个数不可能超过被划分的函数个数，因此我们有 CM 的行数不大于列数：

$$row\_num(CM)\leq col\_num(CM).$$

上式中 row_num 表示行数，col_num 表示列数. 上式只是给出了 CM 行数的上界. 实际情况中的分类数可能远远小于这个上界. 通常情况下，在云环境中，可以假设一个云平台管理员可以管理至少 10 个 RESTful 函数，即一个分类里至少包含 10 个元素. 因此我们可以给 CM 的行数一个更严格的上界. 这样可以减少 CM 的行数，从而减小遗传算法的开销. 需要注意的是，这个上界是针对本文场景的一个性能上的优化，该限制不影响本文方法的适用性.

CM 规则 2：非空列. 由于本方法的目标是把所有 RESTful 函数划分到不同的权限类中，因此每一个 RESTful 函数都至少应该被划分到一个类里，对应到 CM 就是 CM 的每一列都应该非空.

6.3 测试矩阵

类似 OpenStack 的大型分布式软件平台通常都包含多个组件. 集成测试的测试用例包含一系列针对管理任务相关的 RESTful 函数的测试，可以认为测试用例在某种意义上代表了对 RESTful 函数权限的划分. 然而，我们不能将基于测试用例的划分直接作为最终划分结果，有两个原因：(1) 测试用例

的个数通常情况下远比期望划分的权限类个数多，有时甚至超过了待分类的 RESTful 函数个数. 例如，OpenStack 的计算组件 Nova 有 74 个管理 API 函数，但是却有 164 个测试用例；(2) 直接采用测试用例作为划分结果会产生很多权限重叠. 因为测试用例的目的是尽可能覆盖所有使用场景，并不关心测试的函数是否重叠，因此完备的测试集通常会导致大量权限重叠，直接使用这个结果作为权限划分依据会导致大量权限滥用. 然而，由于集成测试体现了从软件开发的角度出发对管理任务的理解，本方法仍然将集成测试的结果作为重要参考. 因此，本文提出了测试矩阵(Test Matrix，TM)的概念. TM 与上文中定义的 CM 类似，唯一的区别在于矩阵行的含义. TM 中的每一行代表了一个测试用例，而不是一个分类. TM 对测试用例之间的顺序不敏感，按照集成测试运行次序定义测试用例顺序即可. TM 中的列和元素的含义与 CM 保持一致. 举例来说，由于 OpenStack Nova 针对 74 个管理 API 函数存在 164 个测试用例，则可以依此构造一个 164×74 的 TM. 由于篇幅的限制，无法展示整个 TM，因此这里从上述 TM 中抽取一些行和列，给出一个 20×12 的 TM 的例子：

$$TM=\begin{pmatrix}
0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&1&0&0&1&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&1\\
0&0&0&0&0&0&0&0&0&0&0&1\\
0&1&1&0&0&0&0&0&0&0&0&0\\
1&0&1&0&0&0&0&0&1&0&0&0\\
1&1&1&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&0&0&0&0&0&1\\
1&1&1&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&0&0&0&0&0&1
\end{pmatrix}.$$

该 TM 列的含义与上一节给出的 CM 例子中的含义一致. 其 20 个行的含义如下：

1. list_servers_by_admin
2. list_servers_filter_by_error_status
3. list_hosts
4. how_host_detail
5. create_list_delete_floating_ip_bulk
6. list_fixed_ip_details
7. aggregate_add_non_exist_host
8. aggregate_list_as_user
9. aggregate_add_host_list
10. aggregate_create_update_with_az
11. create_agent
12. list_agents
13. list_servers_filter_by_exist_host
14. reset_state_server
15. flavor_access_list_with_public_flavor
16. flavor_update_more_key
17. list_non_public_flavor
18. create_server_with_non_public_flavor
19. flavor_access_add_remove
20. resize_server_revert_deleted_flavor

由于一个测试用例所测试的 RESTful 函数集合可以是另一个测试用例所测试的集合的子集，当权限划分满足了后者时，也就满足了前者，因此权限划分方法只要满足后者就可以. 据此我们可以进行优化，将 TM 中所有被其他行对应的测试用例所包含的行去掉，以减少矩阵行数，节省计算开销. 以 OpenStack Nova 为例，通过本文提出的优化方法，TM 的行数从 174 减少到了 108，效果比较明显.

6.4 初始种群

根据遗传算法的原理，需要定义初始种群. 初始种群是由 CM 组成的集合，可以随机产生. CM 的最大行数和列数在上文 6.2 节已经定义. 需要注意的是，如果 CM 中的一行全部为 0，则认为该行对应的分类不存在. CM 随机生成的方式如下：首先 CM 全部元素置 0，然后对每一列，翻转任意一个元素为 1. 这样保证了初始化时，每一个权限都恰好分到一个权限类中.

6.5 变异算子

遗传算法依赖变异算子实现多样性. 针对本文场景，变异需要将原本包含在权限类里的权限去掉，或者添加新的权限，由于 CM 中的元素表示某个权限类是否包含某个权限，因此变异本质上就是考虑如何翻转 CM 中的元素，即多少元素进行 1→0 翻转，多少元素进行 0→1 翻转. 根据遗传算法经典的参数设置，我们假设 CM 中有 10% 的元素进行 1→0

翻转.接下来,在考虑 0→1 翻转时,由于一个 CM 中 0 和 1 出现的频率是不一样的,需要保证元素翻转后,0 和 1 出现的频率维持不变.因此需要首先计算在初始种群中 0 和 1 出现的概率.由于初始种群的 CM 中,每一个权限都恰好分在一个权限类中,因此 CM 的每一列都恰好出现一个元素 1,0 与 1 的比例则为($category\_max-1$)∶1.$category\_max$ 是 CM 的行数上界.综上,为了保证变异时元素 0 和 1 出现的频率保持不变,可以定义变异概率为

$$N_{\text{mutate}}^{1\to 0}=10\%,\ N_{\text{mutate}}^{0\to 1}=\frac{N_{\text{mutate}}^{1\to 0}}{category\_max-1} \quad (2)$$

需要注意的是,由于 CM 中任意一个元素 1 都有可能在变异中被翻转为 0,这有可能导致破坏 CM 规则 2,需要针对这种情形作特殊处理.即在 CM 进行变异后,若发现某列全部为 0,则随机将这列的一个元素翻转为 1,以保证每列都有一个非 0 元素.

6.6　交叉算子

遗传算法中,交叉算子可以实现 CM 个体之间的优势交换,从而产生更强的个体.通过采用经典的随机配对作为配对策略,需要在一个种群中随机选择两个双亲 CM 进行交叉操作,产生新的个体进入下一次迭代.目前主流的交叉算子包括单点交叉、双点交叉、均匀交叉、算数交叉等.本文采用最典型的单点交叉,以最大限度地保存父母个体的优势.在本文场景中,单点交叉就是找到一条分割线对两个 CM 切割后进行交叉.分割线可以为沿 CM 行方向(即纵向)或者沿 CM 列方向(即横向).由于 CM 规则 2 指定了 CM 中每一列都至少有一个非 0 元素,因此沿横向分割并交换容易导致全 0 列的出现.而沿纵向进行分割则不会破坏每列内部的元素,因此不会对 CM 规则 2 造成影响.接下来,利用随机确定的纵向分割线将两个双亲 CM 分为左右两部分,选取第一个双亲 CM 的左部和第二个双亲 CM 的右部,组成新个体的 CM.

6.7　适应度函数

根据上文中提出的权限划分三原则,适应度函数需要满足以下 3 个约束:(1)分类个数应尽可能与用户期望的分类个数接近;(2)分类之间的权限重叠尽可能少;(3)分类应该覆盖尽可能多的测试用例.可以给出如下公式:

$$F(x)=c_1F_1(x)+c_2F_2(x)+c_3F_3(x) \quad (3)$$

在上式中,$F_1(x)$,$F_2(x)$,$F_3(x)$分别是 3 个约束的适应度分数,取值为[0,100].x 表示需要被评估的 CM.c_1,c_2,c_3是 3 个约束的权重,具体取值将在实验部分进行讨论.

(1)分类个数:本文权限划分方法可以接受用户期望的分类个数作为参数,定义为 I_{expected}.本文假设云服务提供商已经有了合适的 I_{expected},如其分配的管理员的人数,以实现每个管理员负责一个权限类.若云服务提供商没有提供期望分类个数,则可以设置 I_{expected}的默认值,或者使 c_1 恒为 0,从而使约束 $F_1(x)$不起作用.本文只讨论存在用户期望分类个数的情形.$F_1(x)$应该在分类个数接近 I_{expected}的取值高,偏离时取值低.因此可以采用最简单的线性函数进行描述:

$$F_1(x)=\max(100-w\times|N_{\text{category}}(x)-I_{\text{expected}}|,0) \quad (4)$$

其中,w 是偏离代价,$w=3$ 的取值就满足本方法要求.

(2)权限重叠个数:权限划分的理想情况是,CM 每一列都恰好有一个元素为 1,即权限类之间没有重叠发生.然而在实际情况中,为了满足测试用例覆盖率的要求,CM 中的每一列通常都会产生多个取值 1,这样每个权限类中才有足够的权限覆盖更多的测试用例.相比 CM 初始化时每列恰好包含一个取值 1 来说,当前 CM 中每多出一个取值 1,则称为一个“权限重叠”.通过对权限重叠计数可以估计权限重叠的严重程度.计算整个 CM 的权限重叠个数时,存在一个优化:算法即将整个 CM 矩阵的元素和,减去 CM 列数,即为 CM 的权限重叠个数:

$$\begin{aligned}&N_{\text{overuse}}(x)=sum(x)-col\_num(x),\\&N_{\text{overuse}}^{\max}(x)=col\_num(x)\times(row\_num(x)-1)\end{aligned} \quad (5)$$

其中,$N_{\text{overuse}}^{\max}(x)$是理论上 $N_{\text{overuse}}(x)$的最大值.当这个最大值成立时,权限重叠最为严重,$F_2(x)$应该取最小值 0,因此可以设计 $F_2(x)$如下:

$$F_2(x)=100\times\left(1-\frac{N_{\text{overuse}}(x)}{N_{\text{overuse}}^{\max}(x)}\right) \quad (6)$$

(3)覆盖测试用例个数:根据上文的阐述,权限分类覆盖了一个测试用例,代表着分类后,管理员仍然可以正常执行测试用例所代表的管理任务.因此权限分类所覆盖的测试用例个数,也展现了该分类是否对原有管理任务具备良好的支持.具体到矩阵层面,就需要去比较 CM 中的每一行与 TM 中的每一行.由于遗传算法的每一代种群都由数量众多的 CM 个体组成,每一个 CM 都需要与 TM 进行这样的比较过程,开销较大.为了提升性能,本文针对覆盖测试用例个数的计算提出了一个基于矩阵乘法的

《计算机学报》

优化算法.记 CM 为 x,TM 为 y,该算法由 3 个步骤组成:

(1)翻转 CM 中所有元素,并转置.用 TM 乘以刚才的结果,再对结果进行翻转(非零值翻转为 0),结果记为 $\boldsymbol{M}_1$.则 $\boldsymbol{M}_1$ 中的元素(i,j)代表是否第 j 个分类覆盖了第 i 个测试用例.

$$\boldsymbol{M}_1=1-y\times(1-x)^{\mathrm{T}} \tag{7}$$

(2)通过使 $\boldsymbol{M}_1$ 乘以一个 $col\_num(\boldsymbol{M}_1)\times 1$ 的全 1 矩阵,对 $\boldsymbol{M}_1$ 的每一行求和.把结果进行二值化(即非零值变为 1),得到矩阵 $\boldsymbol{M}_2$.$\boldsymbol{M}_2$ 是一个行数为测试用例个数,列数为 1 的矩阵,$\boldsymbol{M}_2$ 中的第 i 个元素表示是否被评估的 CM 覆盖了第 i 个测试用例.$\boldsymbol{J}_{m,n}$ 表示 $m\times n$ 的全 1 矩阵.

$$\boldsymbol{M}_2=binarize(\boldsymbol{M}_1\times\boldsymbol{J}_{col\_num(\boldsymbol{M}_1),1}) \tag{8}$$

(3)通过使一个 $1\times row\_num(\boldsymbol{M}_2)$ 的全 1 矩阵乘以 $\boldsymbol{M}_2$,对 $\boldsymbol{M}_1$ 的所有元素求和,得到矩阵 $\boldsymbol{M}_3$.$\boldsymbol{M}_3$ 是一个 1×1 矩阵,其元素值就是 CM 在 TM 中所覆盖的测试用例个数.

$$\boldsymbol{M}_3=\boldsymbol{J}_{1,row\_num(\boldsymbol{M}_2)}\times\boldsymbol{M}_2 \tag{9}$$

最后,记 N_{cases} 为全部测试用例的个数,则 $F_3(x)$可以如下计算:

$$F_3(x)=100\times\frac{\|\boldsymbol{M}_3\|}{N_{\text{cases}}} \tag{10}$$

6.8 选择算子

接下来介绍本方法的选择算子.变异和交叉的概率分别记为 p_m 和 p_c.为了保持种群的稳定,交叉操作后不删除双亲个体,因此种群在交叉后将会产生额外的 $p_c/2$ 的种群.为了选择优势基因,可以在每次迭代后需要淘汰适应度最差的 $p_c/2$ 数量的个体,这样就可以维持每次迭代时的种群规模不变.迭代的停止条件为达到特定的迭代次数.综上,本方案中遗传算法的具体运行步骤为:

1.以随机方式初始化第一代种群;

2.种群中随机选择 p_m 数量的个体进行变异操作;

3.种群中随机选择 p_c 数量的个体两两配对进行交叉操作,这时种群规模会增加 $p_c/2$;

4.以预先定义的适应度函数评估所有个体;

5.淘汰适应度最差的 $p_c/2$ 数量的个体,恢复种群规模.重新执行步骤 2~5,直到触发停止条件.

6.9 创新性分析

传统上,权限划分问题有两种解决方案:自顶向下[13]和自底向上[29].自顶向下的方案需要对现有的组织架构(如企业)进行分析,提取出不同的角色,然后再将角色赋予对应的权限,从而实现权限划分.该方式的难点在于目前没有通用的手段能够从业务组织中直接提取出角色,并且需要大量的人工分析和推导.自底向上的方案则与自顶向下正好相反,其利用目前系统中已有的用户与权限的映射来推导潜在的角色,通常会采用数据挖掘等技术将推导过程自动化,从而分析出权限的分类.然而该方式的不足在于挖掘的准确性受限于训练样本,从而无法保证权限分类的合理性.目前的权限划分技术,通常都借鉴了以上两种方案的思想,并提出自己的改进.

评价权限划分方法的因素主要包括:(1)是否依赖现有用户-权限映射等信息;(2)是否支持 hybrid 特性,即权限分类结果是否可以考虑用户的输入;(3)易于修改的程度,即当分类标准发生变化时,权限划分方法是否能够通过少量修改提供适配;(4)方法自动化的程度,自动化程度越高,表示人工参与越少.在表 1 中,将本文方法与其他具有代表性的方法,如图优化方法、基于场景的方法等进行了比较.与其他方案相比,本文基于遗传算法的方案的创新性在于:(1)不依赖现有的用户-权限映射,只依赖于软件的集成测试集,由于目前的软件开发流程通常都包含集成测试,因此这个依赖是可以接受的;(2)能够接受策略管理员输入的期望分类个数作为参数,从而满足个性化的权限管理需求;(3)由于遗传算法本身是一种基于适应度函数的进化算法,因此通过修改适应度函数,就可以生成满足不同分类标准的权限划分结果,而不需要修改算法的内部细节.相比之下,其他方法大多需要修改方法本身;(4)本文方法自动化程度高,除了输入期望分类个数等参数外(该参数也是可选的),不需要人工参与.相比之下,自顶向下的方法需要人工分析组织结构,而基于场景的方法也需要人工定义符合其规范的场景信息,自动化程度不够高.基于用例的方法以用例文档为输入,与本文的测试用例有相似之处,但是该方法直接将用例文档的需求映射为权限划分结果,缺乏对权限划分原则的考虑,也无法接受用户自定义输入.

表 1 权限划分方法比较

权限划分方法	是否依赖用户权限映射	是否支持 hybrid 特性	易于修改程度	自动化程度
自顶向下[13]	不依赖	支持	低	低
自底向上[29]	依赖	不支持	中	高
图优化方法[29]	依赖	支持	中	高
基于场景方法[24]	不依赖	不支持	低	中
基于用例方法[23]	不依赖	不支持	低	中
本文方法	不依赖	支持	高	高

《计算机学报》

7　实验验证

7.1　原型系统

本文所搭建的 OpenStack 实验云平台配置为：1 个控制节点，2 个网络节点，8 个计算节点，4 个块存储节点. 各节点的硬件配置如下：

控制节点：6 核 Core i7，6 GB 内存，25 GB 磁盘；

网络节点：1 核 Core i7，1 GB 内存，20 GB 磁盘；

计算节点：2 核 Core i7，2 GB 内存，25 GB 磁盘；

块存储节点：1 核 Core i7，1 GB 内存，150 GB 磁盘.

本文利用 RestPL 对 OpenStack 云平台原有的授权机制进行了安全增强，RestPL 强制实施框架的架构如图 1 所示，可以分为请求过滤器和策略服务两部分.

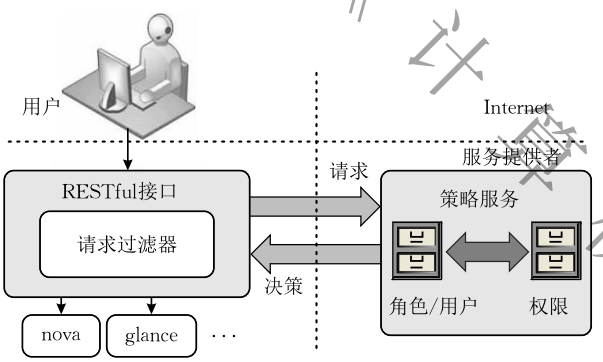


图 1　RestPL 强制实施框架

(1) 请求过滤器：请求过滤器负责过滤和拦截经过 OpenStack RESTful 接口的所有请求. 每一个向云平台发起的 RESTful 请求都会首先经过请求过滤器的过滤. 请求过滤器会将请求的安全上下文(Subject，Object，Verb，Query 等字段)发送到策略服务. 策略服务则会根据 RestPL 策略进行决策，决定是否允许该访问. 若允许，则请求过滤器放行该请求，否则会直接返回 HTTP 403 拒绝访问错误.

(2) 策略服务：策略服务是云平台的一部分，负责 RestPL 策略的决策和存储. 类似于其他组件，策略服务本身也以 RESTful 调用的形式提供策略编辑的接口，方便策略管理员制定、修改 RestPL 策略，以满足其权限管理的需求. 本文中所指的策略管理员既包括云平台管理员，也包括租户管理员. 云平台管理员是云服务提供商的人员，执行云平台级别的权限管理操作，其制定的安全策略对所有租户生效；租户管理员则是一个租户内部的管理者，如企业的 IT 设施负责人，只负责管理租户内部云资源的访问权限，其制定的安全策略只对本租户生效. 本文方法对两类策略管理员都适用，在文中不再进行详细区分.

本文用 Python 实现了 RESTful 权限划分方法，支持对 OpenStack 等主流云平台进行权限划分，其方法流程如图 2 所示. 整个权限划分分为两个阶段：测试阶段和运行阶段. 在测试阶段中，日志收集模块以拦截钩子的形式加载到云平台的 RESTful 接口的请求过滤器中，以实现跟踪记录所有的 RESTful 请求，记录的内容包括当前运行的测试用例名称、该测试用例所包含的 RESTful 请求，这些内容被存储在日志文件中. 然后数据清洗模块会对日志数据进行数据清洗，过滤掉重复的数据，然后由矩阵生成模块处理得到测试矩阵 TM. 在运行阶段，$F(x)$ 生成模块接受策略管理员提供的期望权限分类个数 I_{expected} 作为输入，生成对应的适应度函数 $F(x)$. 遗传算法模块则利用测试矩阵 TM 和适应度函数 $F(x)$，运行遗传算法，计算最优的分类矩阵 CM，从而获得云平台的权限划分结果. 该权限划分结果可以利用 RestPL 的 Role 元素来描述不同的角色，从而制定相应的 RestPL 策略.

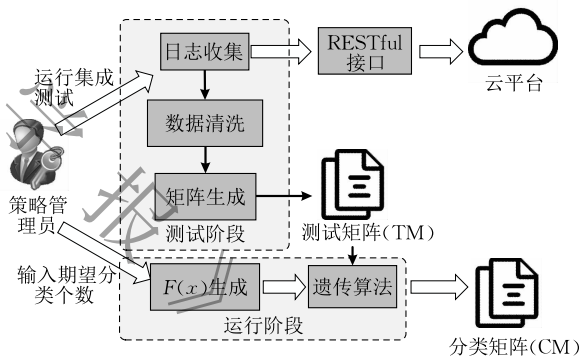


图 2　权限划分方法流程

7.2　策略评估性能

本文以 OpenStack 原有策略(policy.json)、XACML 策略作为对照，对 RestPL 策略评估的性能进行了测试. 测试集为 OpenStack Tempest. Tempest 是专门用来测试 OpenStack RESTful 接口 API 的测试集，包含对计算、镜像、网络、存储等多个组件的测试. 首先在无策略的状态下，运行 Tempest 测试集，利用 5.3 节的策略自动生成算法，生成与测试集对应的 RestPL 策略. 并人工设计出与 OpenStack 原有策略、RestPL 策略语义相近的 XACML 策略. XACML 决策引擎则采用 Sun's XACML 开源库. 利用 Tempest 测试集对 OpenStack 原有策略、XACML 策略、RestPL 测试三种场景进行测试，测试结果如表 2 所示.

《计算机学报》

表 2 Tempest 测试结果 (单位:s)

组件	原有策略开销	XACML 开销	增加开销/%	RestPL 开销	增加开销/%
Nova	651.58	859.18	31.8	527.43	−19.1
Glance	229.70	341.05	48.5	175.91	−23.4
Neutron	230.31	382.50	66.1	168.22	−27.0
Cinder	136.83	199.67	45.9	107.64	−21.3
Heat	292.62	377.19	28.9	246.80	−15.7
Ceilometer	618.79	812.94	31.4	556.23	−10.1

表 2 中的开销专指运行 Tempest 测试集的时间,不包含策略生成的时间.由于在测试场景下,直接采用自动生成的 RestPL 样本策略就可满足基本的访问控制需求,不需要人工进行策略精化,因此开销也不包含策略精化的时间.可以看出,相比 OpenStack 原有策略,采用 RestPL 可使 RESTful 请求的权限评估时间平均降低 19.4%.与之相反,采用 XACML 语言进行访问控制则会增加 42.1%的评估开销.这主要是由于以下两个原因:(1) RestPL 通过 SRF 直接复用 REST 规范中定义的元素(如 Object,Verb 等),简化了请求的解析过程,因而不需要 XACML 语言中的策略信息点这样的机制进行请求元数据的预处理;(2) XACML 严格限制了请求和响应必须为 XML 格式,由于 XML 与 REST 标准并不兼容,因此需要显式的数据格式转化将 RESTful 请求转换为 XML 格式,再交由 XACML 决策引擎处理.

7.3 策略可用性

本节通过对比 XACML 和 RestPL 的策略制定过程,验证 RestPL 语言的可用性.首先考虑 XACML 策略的制定过程:对于初学者来说,首先需要学习 XACML 的基本知识,然而结合具体的访问控制需求人工地编写策略,最后通过构造相应的 RESTful 请求来验证该策略的可用性,若策略组件对请求的处理与策略中指定的效果一致(同为允许或拒绝),则说明该策略有效.否则,则说明策略无法满足高层需求,还需要进一步修改.在企业内部,策略管理员会经常与不同的策略打交道.由于上述策略设计流程严重依赖人工参与,此过程会占用管理员大量时间,并容易导致配置错误.

接下来讨论 RestPL 策略制定的情形.采用 RestPL 策略的管理员可以首先构造一个请求实例,表示将要被访问控制限制的对象.由于 RestPL 是面向请求的策略描述语言,利用 5.3 节提出的策略自动生成算法可以直接将一个 RESTful 请求转化为 RestPL 样本策略.策略管理员可以在该策略的基础上通过加入 RBAC 角色、修改策略 Effect 字段等方式对策略进行进一步精化.

图 3(a)和(b)分别显示了设计 XACML 策略与 RestPL 策略的步骤.深灰色表示人工参与的程度较大.虚线框则表示该步骤可以由程序自动化完成,不需要人工参与.由于 XACML 中的基本概念,如 PolicySet、Policy、Rule 等无法直接与 RESTful 接口中的资源路径、动作等元素建立映射,因此无法通过构造 RESTful 请求的方式,直接生成样本策略.除非人为设计一套 RESTful 请求到 XACML 样本策略的转换规则.然而,该转换的一致性无法保证,转换过程中有可能发生语义损失,并产生一定时间开销.而由于 RestPL 定义的标准请求格式 SRF 与 RestPL 策略共享 Verb、Object 等语言元素,因此通过 SRF 可以直接建立 RESTful 请求到 RestPL 样本策略的映射,从而简化了手工设计策略的步骤,节省了策略设计的开销.

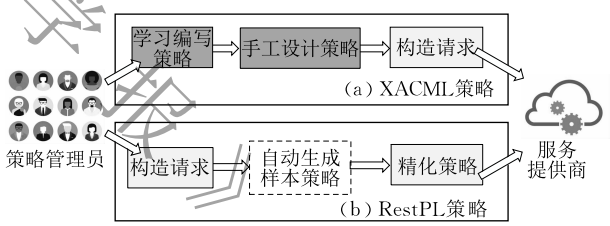
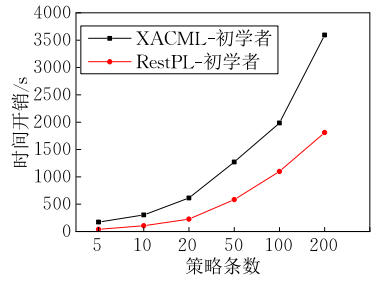
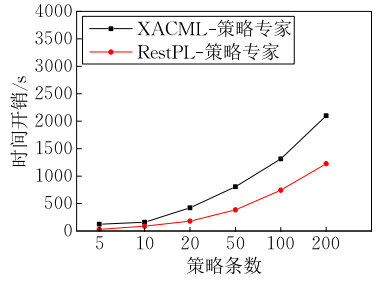


图 3 XACML 策略与 RestPL 策略设计流程的比较

同时,本文以实验方式对比了初学者和策略专家设计策略所花费的时间开销,实验结果如图 4 所示.初学者和策略专家的样本规模各为 20 人.制定的策略以满足 7.2 节中 Tempest 测试集中的测试用例正常运行为需求(即所需权限被策略所授予).如图 4 所示,与 XACML 相比,RestPL 能够减少初学者平均 49.9%的策略设计时间,减少策略专家平均 41.6%的策略设计时间.可以得出,与 XACML 相比 RestPL 更容易节省策略设计的成本,尤其是对于初学者.这是由于 RestPL 针对各种类型的 RESTful 服务都可以表现出相同的语法和相似的语义,因此策略管理员学习掌握的门槛更低.并且 RestPL 能够借助工具实现一定程度的策略自动化生成,从而减少了一部分人工参与.



(a) 初学者



(b) 策略专家

图 4　XACML 策略与 RestPL 策略设计时间开销的比较

7.4　权限划分效果

在本文权限划分算法中，种群规模取值为 160，停止条件为达到 10 000 次迭代. 关于适应度函数的系数取值，由于对于一个权限分类而言，能否满足管理任务的需求，比权限分类个数、权限重叠程度更为重要，因此 $F_3(x)$ 应该赋予比 $F_1(x)$，$F_2(x)$ 更大的权重，据此本文采用如下参数：

$$c_1=1,\ c_2=1,\ c_3=6.$$

关于遗传算法中变异算子和交叉算子的概率，本文采用遗传算法中比较经典的取值：

$$p_m=0.1,\ p_c=0.4.$$

依照本文方法，可以对 OpenStack Nova 的管理性 RESTful 函数进行权限划分，用户期望的分类个数为 5，得到的结果如图 5 所示. 最终得到的最优分类矩阵 CM 的元数据为：

个体适应度：783/800；

覆盖的函数个数：74/74；

权限类个数：5；

权限重叠：18；

覆盖的测试用例个数：108/108.

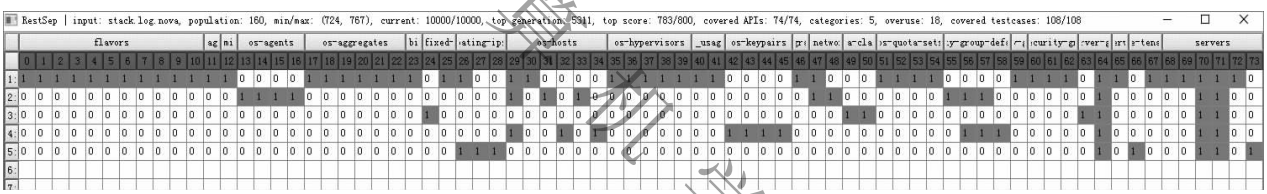


图 5　OpenStack Nova 管理性函数的最优分类矩阵 CM

该元数据表明，该最优 CM 成功地将全部 74 个 Nova 管理函数分成了 5 类. 并且覆盖了全部的 108 个测试用例，即对每一个测试用例，总有一个权限类能够包含执行该用例的全部权限，从而不影响云平台管理任务的运行. 该 CM 只产生了 18 个权限重叠，为了使权限划分覆盖尽可能多的测试用例，而产生权限重叠是不可避免的，只能尽量去减少. 通过观察 5 个权限类所包含的权限，可以发现其不同的特点，可以概括为：

第 1 类. 通用管理员(general)；

第 2 类. 网络管理员(network)；

第 3 类. 配额管理员(quota)；

第 4 类. 秘钥管理员(keypair)；

第 5 类. IP 地址管理员(floating IP).

更进一步分析，本文发现 5 个权限类都包括获取虚拟机(Virtual Machine，VM)的基本信息、删除 VM 等权限. 这是由于 OpenStack 是一个虚拟化云平台，其测试用例执行时通常需要创建临时 VM，针对临时 VM 进行各种操作，然后操作完成后再删除. 然而在实际管理任务中，VM 都是已经正在运行的，并不需要管理员去创建一个 VM 来进行后续操作. 因此这类权限可以认为是噪声，并进行去除. 虽然某些情况下仍存在噪声，但总体上该方法产生的权限分类提供了有益的信息，为接下来利用 RestPL 设计不同的角色，实现最小特权原则提供了有效指导. 例如，以下 RestPL 策略依据上述权限分类结果，将 servers|GET 权限授予通用管理员这个角色：

```
{
  Version: v2,
  Statements: [
    Subject: {
      Domain: admin domain,
      Role: general admin
    },
    Object: servers,
    Verb: GET,
    Effect: Allow
  ]
}
```

除了 OpenStack Nova,本文还测试了 OpenStack 其他组件,如 Glance、Neutron、Cinder、Keystone、Heat 等,结果如表 3 所示.如表 3"分类结果"一列显示,CM 中的每个权限类都根据具体权限的情形,人为地指派了权限类的描述名称.这些名称可能与云平台 API 参考文档中存在的分类有相似之处,但是其实并不相同:文档上给定的分类无法保证覆盖管理任务,其分类个数也是固定的,无法动态调整以符合用户的期望.

表 3 不同测试集的权限分类结果

测试集	测试用例个数	权限个数	权限重叠	$I_{expected}$	分类结果
Nova	108	74	18	5	general, network, quota, keypair, floating IP
Glance	1	7	0	1	general
Neutron	64	63	8	4	general, layer-2, layer-3, quota
Cinder	42	37	5	3	general, QoS&quota, policy
Keystone	131	146	23	7	general, token, credential, domain, group, policy, role&user
Heat	39	57	3	4	general, stack, event, deployment

注:在 CM 中,通常有一个权限类包含了不同类型的 RESTful 权限,没有明显特征,我们称之为通用(general).

7.5 权限划分性能

由于权限划分系统会在云平台运行测试集时进行数据收集,并运行遗传算法求解,因此会产生一定的性能开销.性能开销具体包括测试开销和遗传算法开销两部分,结果如表 4 所示.由于增加了数据收集,测试开销相比原先平均增加了 8.4%左右,遗传算法开销则平均增加了 93.08 s.由于权限划分只在云服务提供商指派管理员时进行,并不是一个频繁发生的操作,因此这样的性能开销是可以接受的.在实际应用中,本文权限划分方法可以直接与云平台的软件构建过程链接起来.每当云服务提供商(也同时是云平台软件的开发者)修改云平台的程序代码后,构建系统都可以自动地调用本文提出的权限划分方法,生成推荐的权限分类,从而对云服务提供商为管理员分配权限提供及时有效的指导.

表 4 权限划分的时间开销 (单位:s)

测试集	原测试时间	加权限划分后测试时间	时间增加/%	遗传算法开销
Nova	643.85	717.59	11.5	116.38
Glance	246.34	270.51	9.8	37.08
Neutron	238.16	249.31	4.7	77.47
Cinder	157.22	165.55	5.3	65.60
Keystone	318.63	340.08	6.7	183.29
Heat	324.35	349.85	7.9	68.80

8 结束语

本文提出了一种新的适用于 RESTful 接口的访问控制策略描述语言 RestPL,该语言采用面向请求的设计思想,可以自动化地根据 RESTful 请求生成对应的样本策略,能够同时表达正向授权和反向授权,支持 RBAC 角色机制,表达能力较强.在应用于不同 Web 服务的 RESTful 接口时,RestPL 可以表现出相同的语法和相似的语义,从而减轻用户学习、编写安全策略的成本.此外,为了解决权限划分的问题,本文提出一种基于遗传算法的云平台 RESTful 权限划分方法,以实现自动化地寻找权限划分的近似最优解.本文首先制定了权限划分的 3 个原则:(1)分类个数与期望相近;(2)权限重叠较少;(3)对原有管理任务的影响较小.进而设计适应度函数、变异算子、交叉算子、选择算子等,同时利用矩阵乘法对适应度计算的性能进行优化.本文在 OpenStack 云平台上实现了 RestPL 强制实施框架,实验结果表明,RestPL 策略评估的性能开销相比 OpenStack 原有策略减少了 19.4%.RestPL 的学习成本与相比 XACML 降低了 41.6%,有效减轻了策略管理员的负担,并且,基于 RestPL 所提出的权限划分方法能有效挖掘出针对 OpenStack 云平台较好的权限划分方案,对原有管理任务影响小,不同分类之间的权限重叠少,并且用户可以提供一个期望的分类个数作为输入,从而满足不同用户的权限划分需求.该方法在性能方面不存在显著的时间开销,从而在不影响服务用户体验的同时,有效细化了云平台权限管理的粒度.

下一步工作包括:研究 RestPL 策略之间的冲突检测问题,给出冲突消解方案;将权限划分方法进一步通用化,研究其在云平台以外的 RESTful 服务中的应用;利用并行遗传算法优化权限划分方法的性能.

致 谢 在此,作者向牛津大学的阮安邦博士对本文工作给予的启发、建议和指导表示感谢!

参 考 文 献

[1] Fielding R T, Taylor R N. Architectural Styles and the Design of Network-Based Software Architectures [Ph. D. dissertation]. University of California, Irvine, USA, 2000

[2] Lin Chuang, Su Wen-Bo, Meng Kun, et al. Cloud computing security: Architecture, mechanism and modeling. Chinese Journal of Computers, 2013, 36(9): 1765-1784(in Chinese)
(林闯，苏文博，孟坤等. 云计算安全：架构、机制与模型评价. 计算机学报，2013，36(9)：1765-1784)

[3] Moses T. Extensible Access Control Markup Language (XACML). Version 2.0. OASIS Standard, Burlington, USA: OASIS, 2005

[4] Ribeiro C, Zuquete A, Ferreira P, et al. SPL: An access control language for security policies with complex constraints //Proceedings of the Network and Distributed System Security Symposium (NDSS 2001). San Diego, USA, 2001: 89-107

[5] Damianou N, Dulay N, Lupu E, et al. The ponder policy specification language//Proceedings of the IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2001). Bristol, UK, 2001: 18-38

[6] Li Ning-Hui, Mitchell J C, Winsborough W H. Design of a role-based trust-management framework//Proceedings of the IEEE Symposium on Security and Privacy (S&P 2002). Berkeley, USA, 2002: 114-130

[7] Han Wei-Li, Lei Chang. A survey on policy languages in network and security management. Computer Networks, 2012, 56(1): 477-489

[8] Liu Qing-Yun, Sha Hong-Zhou, Li Shi-Ming, et al. An access control method based on quantified services and roles in large scale of network visits. Chinese Journal of Computers, 2014, 37(5): 1195-1205(in Chinese)
(刘庆云，沙泓州，李世明等. 一种基于量化用户和服务的大规模网络访问控制方法. 计算机学报，2014，37(5)：1195-1205)

[9] Bertino E, Jajodia S, Samarati P. Supporting multiple access control policies in database systems//Proceedings of the IEEE Symposium on Security and Privacy (S&P 1996). Washington, USA, 1996: 94-107

[10] Minsky N H, Ungureanu V. Unified support for heterogeneous security policies in distributed systems//Proceedings of the 7th USENIX Security Symposium. San Antonio, USA, 1998: 131-142

[11] Ferraolo D, Kuhn R. Role-based access control//Proceedings of the 15th NIST-NCSC National Computer Security Conference. Baltimore, USA, 1992: 554-563

[12] Sandhu R S, Coyne E J, Feinstein H L, et al. Role-based access control models. Computer, 1996, 29(2): 38-47

[13] Coyne E J. Role engineering//Proceedings of the 1st ACM Workshop on Role-Based Access Control (RBAC 1996). New York, USA, 1996: 15-16

[14] Saltzer J H, Schroeder M D. The protection of information in computer systems. Proceedings of the IEEE, 1975, 63(9): 1278-1308

[15] Hu V C, Kuhn D R, Ferraiolo D F. Attribute-based access control. Computer, 2015, 48(2): 85-88

[16] Hachem S, Toninelli A, Pathak A, et al. Policy-based access control in mobile social ecosystems//Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2011). Pisa, Italy, 2011: 57-64

[17] Kagal L, Finin T, Joshi A. A policy language for a pervasive computing environment//Proceedings of the IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003). Lake Como, Italy, 2003: 63-74

[18] Ashley P, Hada S, Karjoth G, et al. Enterprise privacy authorization language (EPAL). IBM Research, 2003, 1(1): 1-70

[19] Jajodia S, Samarati P, Subrahmanian V S. A logical language for expressing authorizations//Proceedings of the IEEE Symposium on Security and Privacy (S&P 1997). Oakland, USA, 1997: 31-42

[20] Roeckle H, Schimpf G, Weidinger R. Process-oriented approach for role-finding to implement role-based security administration in a large industrial organization//Proceedings of the 5th ACM Workshop on Role-Based Access Control (RBAC 2000). Berlin, Germany, 2000: 103-110

[21] Epstein P, Sandhu R. Towards a UML based approach to role engineering//Proceedings of the 4th ACM Workshop on Role-Based Access Control (RBAC 1999). Fairfax, USA, 1999: 135-143

[22] Epstein P, Sandhu R. Engineering of role/permission assignments//Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001). New Orleans, USA, 2001: 127-136

[23] Fernandez E B, Hawkins J C. Determining role rights from use cases//Proceedings of the 2nd ACM Workshop on Role-Based Access Control (RBAC 1997). Fairfax, USA, 1997: 121-125

[24] Neumann G, Strembeck M. A scenario-driven role engineering process for functional RBAC roles//Proceedings of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT 2002). Monterey, USA, 2002: 33-42

[25] Wang Yu-Ding, Yang Jia-Hai, Xu Cong, et al. Survey on access control technologies for cloud computing. Journal of Software, 2015, 26(5): 1129-1150(in Chinese)
(王于丁，杨家海，徐聪等. 云计算访问控制技术研究综述. 软件学报，2015，26(5)：1129-1150)

[26] Richardson L, Ruby S. RESTful Web Services. Sebastopol, USA: O'Reilly, 2008

[27] Abramowitz M, Stegun I A, et al. Handbook of mathematical functions. Applied Mathematics Series, 1966, 55(62): 39

[28] Mitchell M. An Introduction to Genetic Algorithms. Cambridge, USA: MIT Press, 1998

[29] Zhang D, Ramamohanarao K, Ebringer T. Role engineering using graph optimization//Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT 2007). Sophia Antipolis, France, 2007: 139-144

《计算机学报》

LUO Yang, born in 1989, Ph. D. candidate. His research interests include big data and cloud computing security.

SHEN Qing-Ni, born in 1970, Ph. D., professor, Ph. D. supervisor. Her research interests include operating system security, big data and cloud computing security.

WU Zhong-Hai, born in 1968, Ph. D., professor, Ph. D. supervisor. His research interests include big data and cloud computing security, distributed machine learning system.

Background

At present, a growing number of web applications especially cloud computing systems employ representational state transfer (REST) API as the interface to expose their services for simplicity and clarity. For security purposes, service providers prefer to control the access to the provided interface based on the principle of least privilege. However, there is no standard for the access control policy language for RESTful interfaces. And how to divide the administrative privileges remains a difficulty in practice. The authors propose an access control policy description language called RestPL by defining the standard request form and the grammar elements for the language. The authors also attempt to simplify the privilege partitioning problem into a classification problem of RESTful functions, so the permission to call a category of functions can be granted to a specific administrator. This paper proposes a RESTful API classification approach based on genetic algorithm for RestPL. A classification is represented as a 2-dimensional matrix, which is used as the chromosome. Customized operators of selection, mutation and crossover are designed. The fitness function is designed to balance parameters such as number of categories, test case coverage, function overlapping, etc. As far as the authors know, this is the first work that defines an authorization language for RESTful interfaces which can be used to partition the privileges.

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61232005, 61672062 and the National High Technology Research and Development Program (863 Program) of China under Grant No. 2015AA016009.

The research group has long been dedicated to research on the theory and techniques of cloud security including the access control policy languages and the authorization enforcement mechanisms. They construct a security enhanced IaaS cloud platform which is based upon the open-source OpenStack cloud. This platform is scalable and supports various defensive measures such as secure load-balancing, secure virtual machine migration, virtualized trusted computing, etc. This platform offers technical support for modeling and verification of RestPL language of this research.

《计算机学报》