

一种 XACML 规则冲突及冗余分析方法

王雅哲 冯登国

(中国科学院软件研究所信息安全国家重点实验室 北京 100190)
(信息安全共性技术国家工程研究中心 北京 100190)

摘 要 基于属性的声明式策略语言 XACML 表达能力丰富,满足开放式环境下资源访问管理的复杂安全需求,但其自身缺乏对规则冲突检测、规则冗余分析的支持.文中利用规则状态思想描述分析了属性层次操作关联带来的多种冲突类型,在资源语义树策略索引基础上利用状态相关性给出规则冲突检测算法;利用状态覆盖思想分析造成规则冗余的原因,给出在不同规则评估合并算法下的冗余判定定理.仿真实验首先分析了冲突检测算法的运行效率;然后针对多种策略判定系统,验证了基于语义树的策略索引和冗余规则处理可以显著提高判定性能.

关键词 访问控制;规则状态;属性层次;规则冲突检测;规则冗余;XACML

中图法分类号 TP309 **DOI号**: 10.3724/SP.J.1016.2009.00516

A Conflict and Redundancy Analysis Method for XACML Rules

WANG Ya-Zhe FENG Deng-Guo

(State Key Laboratory of Information Security, Institute of Software, Chinese Academy of Sciences, Beijing 100190)
(National Engineering Research Center of Information Security, Beijing 100190)

Abstract XACML is a kind of declarative policy language which has flexible expressive functions based on attributes and satisfies complex security requirements of access management in the open environment, but it lacks the capabilities of detecting conflict rules and analyzing rule redundancy. This paper proposes rule state concept and applies it to analyze several categories of rule conflict caused by attribute hierarchy. In order to detecting and locating these conflicts, resource semantic tree and state relativity are utilized for depicting conflict detecting algorithms. Besides that, rule redundancy is the other issue in this paper. Employing state covering method, the mechanism of rule redundancy is explained, and redundancy judgment theorems are proven for various rules combining algorithms. The emulation tests in the last part of this paper firstly analyze the algorithm's efficiency, secondly indicate that evaluation performance can profit from resource semantic tree index and redundancy disposing.

Keywords access control; rule state; attribute hierarchy; rule conflict detecting; rule redundancy; XACML

1 引 言

通过安全策略描述系统高层安全需求是目前实

现授权访问控制的主要途径.策略就是“一套由管理需求而来的、相对持久的、说明性的规则,此规则约束了系统做出决策的过程”^[1].随着组织间业务协作、分布式计算、跨域资源共享等新一代互联网体系

结构的大量涌现,基于策略管理(policy-based management)面临着新的挑战。开放式计算环境下,跨组织边界的用户和服务资源动态加入或撤出组织,安全策略需要考虑的安全属性种类繁多,授权决策的辅助参数也趋于复杂,组织内授权状态灵活多变,这些情况不可避免导致了内部策略冲突的威胁发生。安全管理初衷能否准确执行、策略间的冲突能否合理消解是授权策略判定系统的难点。

OASIS 发布的访问控制策略标记语言 XACML^[2] 基于属性匹配机制定义策略规则,适用于新一代互联网体系结构下分布式异构访问控制、多管理域策略合成等应用场合,可通过多种属性类型的细粒度刻画定义授权约束,但同时也容易导致比较复杂的策略规则冲突类型。多规则适用于相同访问请求,但判定结果互斥是 XACML 策略冲突的基本类型;属性层次导致的权限继承和权限蕴含也可能引发规则冲突。虽然 XACML 标准中给出了若干冲突消解算法,从判定结果角度规避了策略冲突对访问请求的影响,但其未能从管理视图角度分析造成冲突的细节原因,自身缺乏推理验证的能力。

基于上述原因,本文利用规则状态思想分析了主体属性层次和资源属性层次操作关联导致的多种冲突类型,基于资源属性语义树建立 XACML 策略索引避免了传统遍历式策略检索,提高了策略分析效率。在此基础上分别给出基于属性层次操作关联的冲突检测算法和基于状态相关性的其他类型冲突检测算法,实现对规则冲突的预定位。利用状态覆盖思想分析造成规则冗余的原因,给出不同规则评估合并算法下的冗余判定定理。通过仿真实验表明高效策略索引和冗余规则处理可以显著提高判定性能。

2 相关工作

目前主要的策略描述语言大致归为高层声明式语言和基于逻辑的形式化语言。以 XACML 为典型代表的声明式语言能够面向用户清晰表达系统安全需求,描述能力丰富,但自身缺乏完善的策略分析推理能力,不易发现策略内部的安全漏洞和冲突威胁。类似 ASL^[3-4] 的逻辑类策略语言结构关系严谨,推导能力强,便于用户对整个策略空间建立完整安全视图,但其从高层逻辑到实施机制的转化代价巨大,缺乏应用层支持。

无论是声明式安全策略还是基于逻辑描述的安全策略,冲突检测都是策略分析关注的重点问题。面

向应用的策略冲突消解最早在人工智能和数据库系统等领域引起过人们的注意,其后在授权访问方面得到系统的研究。依据授权判定类型检测冲突是访问控制中最普适的一种策略检测形式。Lupu 提出的模态冲突模型^[5] 根据模态符号将策略冲突定义为 3 种类型:正向授权/负向授权冲突、正向职责/负向职责冲突、正向职责/负向授权冲突,借助模态符号冲突分析对安全策略进行静态检测。有一些研究工作专门就基于角色访问控制模型 RBAC 模型展开;文献[6]将安全策略理解为对系统内实体实施的约束,将约束划分为主客体间访问约束和主体间管理约束,分析了角色分配带来的权限冲突,并根据角色优先级进行冲突消解;文献[7]给出基于系统运行时序的动态冲突检测算法,但只分析了角色内权限冲突的情况,对其他较复杂的冲突类型没有给出具体检测消解方案。

随着 XACML 策略广泛被使用并成为工业界标准,很多研究工作开始注重对其进行分析、推理和验证^[8-10]。Zhang 等^[9-10] 基于 RW 语言验证策略是否满足用户合法需求以及能否阻止非法访问,并将 RW 验证后的策略转化为 XACML 形式。针对多组织场景下存在的策略合并问题,Hughes 等^[11-12] 应用一阶逻辑分析工具 Alloy 对 XACML 策略进行建模,分析验证策略合并后是否与原始策略的安全初衷保持一致性,通过可满足性(SAT)求解器检测预期之外的策略合并结果。Bryans^[13] 利用进程代数分析访问控制策略,将 XACML 协议栈在通信顺序进程 CSP 体系下建模,通过 CSP 语义验证策略特征并进行策略比较。Margave^[14] 是 Fisler 等基于 MT-BDDS(Multi-Terminal Binary Decision Diagrams)开发的策略分析工具,其主要从两方面对策略进行分析:(1)验证某策略是否满足给定的安全特性,可以将这种安全特性理解为用更高层的自然语言描述的安全需求;(2)给定修改前后的某样本策略,分析策略变更对访问请求判定结果的影响。Martin 等^[15-17] 在 Margave 基础上借鉴软件测试的方法,生成变异测试用例检测策略有效性,利用覆盖率量化检测效率指标。

可以看出,上述研究工作主要侧重于策略目标的可满足性验证,借助其他理论方法验证 XACML 策略是否满足特定的系统安全需求和安全限制,未从策略冲突角度对 XACML 内部机制进行深入研究。造成这种状况的一个显著原因是 XACML 标准已经提供了若干冲突消解算法,无论是对资源访问

者还是安全策略制定者都屏蔽了产生策略冲突的过程和原因,针对 XACML 的策略分析工作通过使用消解算法忽略了对策略内规则关系的分析,只关注策略最终给出的判定结果.从提高系统评估效率和降低用户理解专业知识门槛的角度出发,预先定义消解算法是一种有效的解决方案,但是消解算法的目的只是保证访问控制系统评估的可确定性,并没有能力对 XACML 体系结构下的冲突类型进行准确分类和细致刻画.由于 XACML 基于属性集合描述权限结构,因此主体属性层次和资源属性层次间操作关联导致的授权结果互斥是 XACML 策略内规则冲突的主要原因,另外规则间属性集合的重叠也可能造成访问评估的结果互斥. Mazzoleni^[18]等提出了一套较完整的规则间关系定义,用于比较组织间策略的相似性,从而实现一种域间策略自动整合算法.这是在规则分析方面为数不多的研究成果,但其没有对规则间冲突检测进行分析,更没有进一步地定义冲突类型.事实上,通过对策略内规则冲突的预分析和冲突类型分类,安全管理员可提前发现冲突原因,通过调整规则位置、修改属性取值范围合理规避冲突的发生,对无法避免的规则冲突也可以分析其冲突类型,有助于选用更合适的消解算法与安全管理初衷保持一致.开放式应用环境会引发策略规模激增,策略库可能包含大量对访问请求不产生任何判定作用的规则条目,从而降低策略评估匹配效率,因此规则冗余也是策略系统应该关注的问题.到目前为止,仅 Kolovski^[19]等对 XACML 策略中的规则冗余进行了相关研究,其利用描述逻辑(description logics, DL)形式化 XACML 策略,借助已有的 DL 验证分析工具检测策略中的冗余规则,文章还指出,文献[4]中的策略变更影响分析工具也可以用来分析冗余规则,因为针对冗余规则的策略变更不会改变原有策略的判定结果.从本质上讲,XACML 中的冗余规则是针对规则评估合并算法而言的,因此不同合并算法下的冗余规则判断也不尽相同.现有研究工作未从理论分析的角度对规则冗余原因、冗余判定依据等给出清晰定义和论证步骤,只是通过比较直观的方式进行判断分析.

XACML 策略分析以及评估实施都依赖于属性约束匹配的满足性判定,因此策略检索空间的大小直接影响系统运行效率.如何建立高效的策略索引结构是匹配性能提升的重要瓶颈,但现有大多数基于 XACML 实现的访问控制系统并没有在策略索引优化方面取得显著进展.eXist 数据库^①项目中,判定系统针对 Xquery 查询请求采取策略库遍历的

方式进行匹配运算. Sun XACML 系统开发包^②中只提供了一个基于文件策略模式的判定器实例,没有实现复杂的策略匹配索引结构. Parthenon Computing^③实现的策略评估引擎只支持单一策略的评估,不支持对策略库的搜索判定. PRIMA^[20]考虑了 XACML 策略在策略管理点和判定实施点间的安全传输,但没有策略匹配的优化方案. Melcoe PDP^④将策略存储在 XML 类型数据库中,根据给定的属性匹配列表缩减策略检索空间,但其检索优化方案依赖数据库自身处理 XML 的效率优势并且降低了 XACML 的灵活性.环境单一的小规模策略应用场景下,逐条匹配对策略检索效率并没有显著影响,但在大规模策略集成应用中,对访问请求真正实施影响的几条策略可能分布在数以千计甚至万计的策略条目中,遍历匹配的模式会严重降低系统检索有效策略的概率.

3 XACML 特征

本节首先给出规则状态定义,然后利用规则状态分析 XACML 中的权限继承规则和权限蕴含规则,最后分析了规则状态间的相关性问题.

3.1 规则状态

XACML 采用分层嵌套模式定义安全策略. PolicySet 和 Policy 是外部结构化策略容器:PolicySet 容器可以包含若干 Policy 或其他 PolicySet; Policy 容器由若干规则 Rule 组成.策略容器和规则可以分别定义目标元素 Target 从而约束各自的适用范围. Rule 中 effect 元素表示该规则为正向权限(permit)或负向权限(deny),condition 元素可以进一步约束规则适用性(本文忽略 condition 的影响).策略决策点(Policy Decision Point, PDP)将访问请求中各种属性信息和策略容器中目标元素逐层匹配推导出适用规则的判定断言.针对推导出的多条判定断言,XACML 利用规则/策略合并算法计算最终的评估结果.

目标元素定义了策略或者规则适用的主体、资源、动作和环境集合,其主要作用是为 PDP 提供属性匹配规则.实际应用中,外部策略容器的 Target 元素可以定义为内部策略容器和所包含规则各自

① <http://exist.sourceforge.net/xacml.html>

② <http://sunxacml.sourceforge.net>

③ http://www.parthcomp.com/xacml_toolkit.html

④ <http://www.muradora.org>

Target 的并集. 在这种情况下, 外部组件的 Target 元素适用于至少满足一个内部组件 Target 元素的访问请求.

规则元素是推导断言结论的信息来源, 是最小单位的完整授权约束原语, 以下是其形式化描述.

定义 1. 规则状态. sub_i 、 res_i 、 ac_i 和 en_i 分别代表 Rule 中一个主体、资源、动作和环境属性分量, 每个分量包含一组使断言合取式 $\bigcap_{j=1}^m match_j$ 为真的属性值, 其中断言 $match_j$ 是作用在属性值上的布尔函数, $\bigcup_{i=1}^{n_1} sub_i$ 、 $\bigcup_{i=1}^{n_2} res_i$ 、 $\bigcup_{i=1}^{n_3} ac_i$ 、 $\bigcup_{i=1}^{n_4} en_i$ 分别表示各种属性分量的并集, $Decision$ 表示 Rule 的 effect 类型, 则 Rule 的规则状态表示为 $State(\bigcup_{i=1}^{n_1} sub(\bigcap_{j=1}^{m_1} match_j)_i, \bigcup_{i=1}^{n_2} res(\bigcap_{j=1}^{m_2} match_j)_i, \bigcup_{i=1}^{n_3} ac(\bigcap_{j=1}^{m_3} match_j)_i, \bigcup_{i=1}^{n_4} en(\bigcap_{j=1}^{m_4} match_j)_i) \mapsto Decision$.

定义 2. 规则状态满足. 若规则 $Rule_i$ 声明主体 SA 在环境状态为 EA 时, 可以对资源 RA 进行访问动作 AA , 则属性状态向量 (SA, RA, AA, EA) 满足规则状态 $State_i$, 表示为 $State_i \models (SA, RA, AA, EA)^{effect}$.

3.2 权限规则

XACML 规范支持 RBAC 模型中的权限继承机制, 采用策略引用的方式描述角色间的层次关系 (如图 1). 一个角色属性对应一个 PRS 策略和 PPS

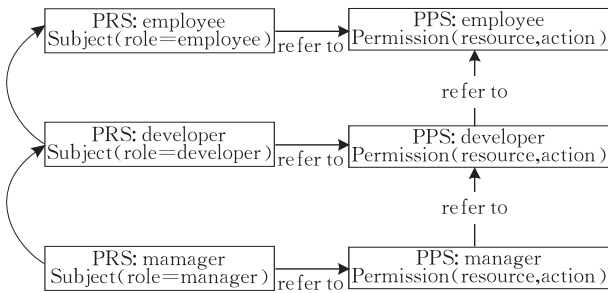


图 1 XACML 中的权限继承

策略, PRS 的目标元素仅定义主体角色属性并且 PRS 引用 PPS 策略, PPS 策略的目标元素通过资源属性和动作属性定义该角色拥有的权限. 若角色 R_2 的 PPS 策略引用了角色 R_1 的 PPS 策略, 表示 R_2 继承 R_1 的权限. 借鉴这种方式对 XACML 进行扩展, 使得其他存在层次关系的主体属性也可以通过类似策略引用的方式描述权限继承. 主体属性间的层次关系蕴含权限间的包含关系, 若上层主体属性针对某资源拥有任意权限, 则该权限沿主体属性层次向

下层继承属性传播. 同样的, 资源属性间也会存在层次关系, 例如 URL 地址、系统资源目录、XML 文档等都具有自然语义结构的资源标识符. 但是通常情况下, 上层资源属性代表粗粒度低敏感的资源内容, 下层资源属性代表细粒度高敏感的资源内容. 考虑信息保护并根据最小权限原则, 资源属性的层次关系隐含了资源受保护的安全级别和敏感性: 上层资源属性对应的正向权限不会蕴含下层资源属性, 下层资源应具有明确正向授权才可以访问; 若系统将负向权限作用于上层粗粒度资源, 则负向权限蕴含到下层细粒度敏感资源. 针对主体属性 SA 、资源属性 RA 、动作属性 AA , 若 SA_1 是 SA_2 的上层属性, 表示为 $SA_2 \triangleleft SA_1$; 若 RA_1 是 RA_2 的上层属性, 表示为 $RA_2 \triangleleft RA_1$. 可如下形式化定义权限规则.

权限继承规则. 当 $SA_2 \triangleleft SA_1$: 若 $State_i \models (SA_1, RA, AA)^{permit}$, 则 $State_i \models (SA_2, RA, AA)^{permit}$; 若 $State_i \models (SA_1, RA, AA)^{deny}$, 则 $State_i \models (SA_2, RA, AA)^{deny}$.

例如角色属性 `developer` 可以继承来自 `employee` 的正向权限 (允许读系统说明文档) 和负向权限 (禁止写系统测试文档).

权限蕴含规则. 当 $RA_2 \triangleleft RA_1$: 若 $State_i \models (SA, RA_1, AA)^{permit}$, 则不能推导出 $State_i \models (SA, RA_2, AA)^{permit}$; 若 $State_i \models (SA, RA_1, AA)^{deny}$, 则 $State_i \models (SA, RA_2, AA)^{deny}$.

例如允许 `employee` 读资源“Code/”并不能推导出允许其读资源“Code/JavaCode”也成立, 禁止 `employee` 写资源“Code/”可以推导出禁止其写资源“Code/JavaCode”.

3.3 状态相关性

策略分析时需要比较规则间关系, 将其转化为规则状态比较, 有助于分析规则间可能存在的判定冲突. 假设 $Rule_i$ 和 $Rule_j$ 的规则状态分别

为 $State_i(\bigcup_{n=1}^{n_1} sub_{i_n}, \bigcup_{n=1}^{n_2} res_{i_n}, \bigcup_{n=1}^{n_3} ac_{i_n}, \bigcup_{n=1}^{n_4} en_{i_n})$ 和 $State_j(\bigcup_{m=1}^{m_1} sub_{j_m}, \bigcup_{m=1}^{m_2} res_{j_m}, \bigcup_{m=1}^{m_3} ac_{j_m}, \bigcup_{m=1}^{m_4} en_{j_m})$, 下面分别给出状态覆盖、状态相交和状态无关的定义.

定义 3. 状态覆盖. 存在关系 $(\forall sub_{i_n}, \exists sub_{j_m} : sub_{i_n} \in \bigcup_{m=1}^{m_1} sub_{j_m} \vee sub_{j_m} \triangleleft sub_{i_n}) \wedge (\forall res_{i_n}, \exists res_{j_m} : res_{i_n} \in \bigcup_{m=1}^{m_2} res_{j_m} \vee res_{i_n} \triangleleft res_{j_m}) \wedge (\forall ac_{i_n} : ac_{i_n} \in \bigcup_{m=1}^{m_3} ac_{j_m}) \wedge (\forall en_{i_n} : en_{i_n} \in \bigcup_{m=1}^{m_4} en_{j_m})$, 则称 $State_j$ 覆盖

$State_i$, 表示为 $State_i < State_j$.

若规则 $Rule_i$ 目标元素中定义的各种属性类型和属性取值范围或者包含在 $Rule_j$ 中(表示为 $sub_{i_n} \in \bigcup_{j=1}^{m_1} sub_{j_m}$, $res_{i_n} \in \bigcup_{j=1}^{m_2} res_{j_m}$ 等), 或者存在 $sub_{j_m} \triangleleft sub_{i_n}$ 和 $res_{i_n} \triangleleft res_{j_m}$ 的情况, 则 $State_i < State_j$. 当 $sub_{j_m} \triangleleft sub_{i_n}$ 时, 下层主体属性 sub_{j_m} 继承上层主体属性 sub_{i_n} 的访问权限, 从权限包含的角度可以认为 $State_j$ 覆盖 $State_i$. 当 $res_{i_n} \triangleleft res_{j_m}$ 时, 将上层属性 res_{j_m} 理解为下层属性 res_{i_n} 的祖先节点, 从资源划分的角度可以认为 $State_j$ 覆盖 $State_i$.

定义 4. 状态相交. 存在关系 $(\bigcup_{n=1}^{n_1} sub_{i_n} \cap \bigcup_{m=1}^{m_1} sub_{j_m} \neq \emptyset) \vee (\bigcup_{n=1}^{n_2} res_{i_n} \cap \bigcup_{m=1}^{m_2} res_{j_m} \neq \emptyset) \vee (\bigcup_{n=1}^{n_3} ac_{i_n} \cap \bigcup_{m=1}^{m_3} ac_{j_m} \neq \emptyset) \vee (\bigcup_{n=1}^{n_4} en_{i_n} \cap \bigcup_{m=1}^{m_4} en_{j_m} \neq \emptyset)$ 且 $State_j$ 和 $State_i$ 间不存在覆盖关系, 则称 $State_j$ 和 $State_i$ 状态相交, 表示为 $State_i \xleftrightarrow{granularity} State_j$.

目标元素 $Target_i$ 和 $Target_j$ 如果存在部分重合, 则两条规则可能在某些情况下适用于同一请求. 根据目标元素的结构, 会存在不同层次和粒度的重合相交关系: 可能是某个属性分量的取值范围存在交集, 也可能是属性分量组合的取值范围存在交集.

$granularity$ 表示状态相交的粒度, 例如 $Rule_i$ 的主体属性为 $sub_i \{role[developer, tester], group[internal, external] \wedge service-age[3 \sim 5]\}$, $Rule_j$ 主体属性为 $sub_j \{role[developer, designer], group[internal] \wedge service-age[2 \sim 4]\}$, 则 $granularity$ 表示为 $(sub.role, sub.group \wedge sub.service-age)$. 前者表示针对 $role$ 属性存在交集 ($role.developer$), 后者针对 $group$ 属性和 $service-age$ 属性组合存在交集 ($group.internal \wedge service-age[3 \sim 4]$).

定义 5. 状态无关. 存在关系 $(\forall sub_{i_n} : sub_{i_n} \notin \bigcup_{m=1}^{m_1} sub_{j_m}) \wedge (\forall res_{i_n} : res_{i_n} \notin \bigcup_{m=1}^{m_2} res_{j_m}) \wedge (\forall ac_{i_n} : ac_{i_n} \notin \bigcup_{m=1}^{m_3} ac_{j_m}) \wedge (\forall en_{i_n} : en_{i_n} \notin \bigcup_{m=1}^{m_4} en_{j_m})$, 则称 $State_j$ 和 $State_i$ 状态无关, 表示为 $State_i \perp State_j$.

如果两条规则间的各种属性分量集合以及具有相同属性名称的属性分量的取值范围完全不同, 则认为两规则状态间状态无关.

4 XACML 规则冲突检测分析

本节首先分析属性层次互操作引发的冲突类

型, 然后利用基于资源语义树的策略索引结构, 分别给出基于属性层次操作关联的冲突检测算法和基于状态相关性的其他类型冲突检测算法.

4.1 属性层次操作关联类型分析

虽然 XACML 本身提供了允许优先 (permit-override)、拒绝优先 (deny-override)、首次适用 (first-applicable) 等合并算法对同一访问请求产生的不同评估结果进行冲突消解以期获得唯一判定结果, 但只是从判定结果角度规避了策略冲突对访问请求的影响, 缺乏从管理视图角度分析造成冲突的细节原因以及冲突预定位能力.

权限继承和权限蕴含是 XACML 策略冲突的主要原因之一. 例如角色 *manager* 的权限策略中包含规则状态 $State(code, write) \mapsto deny$, *developer* 角色的权限策略中包含规则状态 $State(code, write) \mapsto permit$, *manager* 自身权限和继承自 *developer* 的权限发生冲突; 规则状态 $State(group, extenal, code, read) \mapsto deny$ 和 $State(group, extenal, code/java code, read) \mapsto permit$ 针对下层资源 “code/java code” 发生冲突. 当比较两条具体的访问规则时, 可以把存在直接或间接层次关系的属性关系归纳为二元关系, 其基本操作关联如图 2(a) 所示 (设定动作属性相同, 暂不考虑环境属性): 主体属性层次关系中 SA_1 表示上层属性, SA_2 表示下属性, SA_2 继承来自 SA_1 的权限; 资源属性层次关系中 RA_1 表示上层粗粒度资源, RA_2 表示下层细粒度资源, RA_1 的负向权限可蕴含至 RA_2 . 图 2(i)~(iv) 描述了两条规则既存在主体属性层次也存在资源属性层次的情况, 此时属性层次关联存在两种可能: 交叉 (同一规则内定义的主体属性和资源属性相对另一规则位于不同层次) 和同级 (同一规则内定义的主体属性和资源属性相对另一规则位于相同层次), 结合两种 effect 结果, 组合为 4 种类型; 图 2(v)~(vi) 描述了只存在资源属性层次的情况, 此时两条规则定义了相同的主体属性, 根据不同的 effect 结果分为两种类型; 图 2(vii)~(viii) 描述了只存在主体属性层次的情况, 此时两条规则定义了相同的资源属性, 根据不同的 effect 结果分为两种类型. 以上类型都是规则间属性层次操作关联的原子类型, 囊括了属性层次导致的各种规则冲突原因, 其它由属性层次引起的更复杂的冲突场景都是这些原子类型的组合类型, 例如图 2(b) 中描述的冲突场景可以分解成虚线所指的 6 种原子类型.

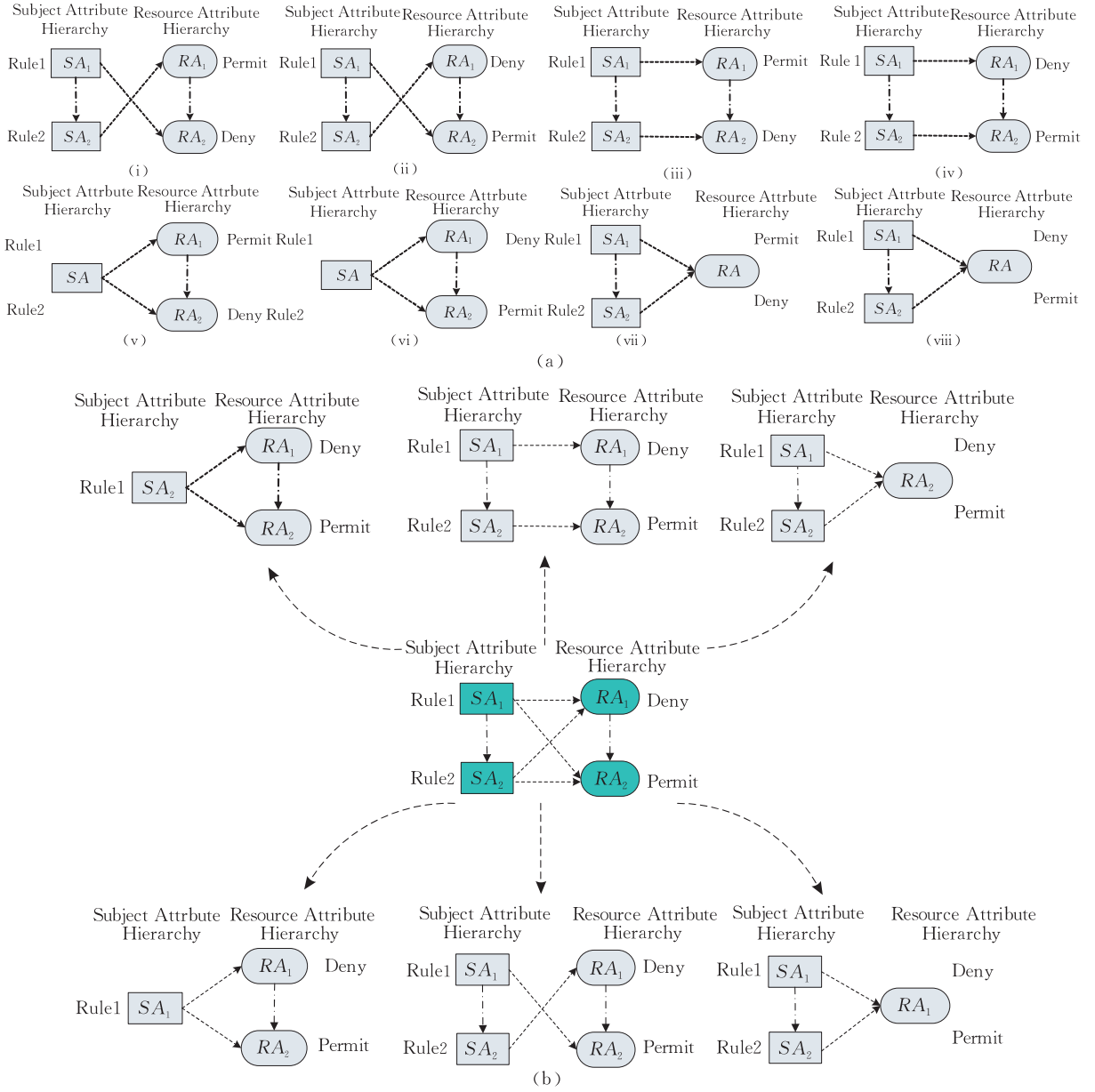


图 2 主体层次和层次操作关联类型

要说明的是,并不是所有的原子类型都导致实际的规则冲突.若 Rule1 的规则状态 $State_1$ 和 Rule2 的规则状态 $State_2$ 针对共同适用的访问请求

存在判定冲突,表示为 $conflict_{State_1}^{State_2}$,下面对各种属性层次关联的原子类型进行冲突判定的形式化推导 ($SA_2 \triangleleft SA_1, RA_2 \triangleleft RA_1$):

$$(i) \quad State_1 \models (SA_1, RA_2, AA)^{Deny}, State_2 \models (SA_2, RA_1, AA)^{Permit}$$

$$\left\{ \begin{array}{l} State_1 \models (SA_1, RA_2, AA)^{Deny} \Rightarrow State_1 \models (SA_2, RA_2, AA)^{Deny} \\ State_2 \models (SA_2, RA_1, AA)^{Permit} \not\Rightarrow State_2 \models (SA_2, RA_2, AA)^{Permit} \end{array} \right\} \not\Rightarrow conflict_{State_2}^{State_1}$$

$$(ii) \quad State_1 \models (SA_1, RA_2, AA)^{Permit}, State_2 \models (SA_2, RA_1, AA)^{Deny}$$

$$\left\{ \begin{array}{l} State_1 \models (SA_1, RA_2, AA)^{Permit} \Rightarrow State_1 \models (SA_2, RA_2, AA)^{Permit} \\ State_2 \models (SA_2, RA_1, AA)^{Deny} \Rightarrow State_2 \models (SA_2, RA_2, AA)^{Deny} \end{array} \right\} \Rightarrow conflict_{State_2}^{State_1}$$

$$(iii) \quad State_1 \models (SA_1, RA_1, AA)^{Permit}, State_2 \models (SA_2, RA_2, AA)^{Deny}$$

$$\left\{ \begin{array}{l} State_1 \models (SA_1, RA_1, AA)^{Permit} \Rightarrow State_1 \models (SA_2, RA_1, AA)^{Permit} \not\Rightarrow State_1 \models (SA_2, RA_2, AA)^{Permit} \\ State_2 \models (SA_2, RA_2, AA)^{Deny} \Rightarrow State_2 \models (SA_2, RA_2, AA)^{Deny} \end{array} \right\} \not\Rightarrow$$

$conflict_{State_2}^{State_1}$

- (iv) $State_1 \models (SA_1, RA_1, AA)^{Deny}, State_2 \models (SA_2, RA_2, AA)^{Permit}$
 $\left\{ \begin{array}{l} State_1 \models (SA_1, RA_1, AA)^{Deny} \Rightarrow State_1 \models (SA_2, RA_1, AA)^{Deny} \Rightarrow State_1 \models (SA_2, RA_2, AA)^{Deny} \\ State_2 \models (SA_2, RA_2, AA)^{Permit} \Rightarrow State_2 \models (SA_2, RA_2, AA)^{Permit} \end{array} \right\} \Rightarrow conflict_{State_2}^{State_1}$
- (v) $State_1 \models (SA, RA_1, AA)^{Permit}, State_2 \models (SA, RA_2, AA)^{Deny}$
 $\left\{ \begin{array}{l} State_1 \models (SA, RA_1, AA)^{Permit} \not\Rightarrow State_1 \models (SA, RA_2, AA)^{Permit} \\ State_2 \models (SA, RA_2, AA)^{Deny} \Rightarrow State_2 \models (SA, RA_2, AA)^{Deny} \end{array} \right\} \not\Rightarrow conflict_{State_2}^{State_1}$
- (vi) $State_1 \models (SA, RA_1, AA)^{Deny}, State_2 \models (SA, RA_2, AA)^{Permit}$
 $\left\{ \begin{array}{l} State_1 \models (SA, RA_1, AA)^{Deny} \Rightarrow State_1 \models (SA, RA_2, AA)^{Deny} \\ State_2 \models (SA, RA_2, AA)^{Permit} \Rightarrow State_2 \models (SA, RA_2, AA)^{Permit} \end{array} \right\} \Rightarrow conflict_{State_2}^{State_1}$
- (vii) $State_1 \models (SA_1, RA, AA)^{Permit}, State_2 \models (SA_2, RA, AA)^{Deny}$
 $\left\{ \begin{array}{l} State_1 \models (SA_1, RA, AA)^{Permit} \Rightarrow State_1 \models (SA_2, RA, AA)^{Permit} \\ State_2 \models (SA_2, RA, AA)^{Deny} \Rightarrow State_2 \models (SA_2, RA, AA)^{Deny} \end{array} \right\} \Rightarrow conflict_{State_2}^{State_1}$
- (viii) $State_1 \models (SA_1, RA, AA)^{Deny}, State_2 \models (SA_2, RA, AA)^{Permit}$
 $\left\{ \begin{array}{l} State_1 \models (SA_1, RA, AA)^{Deny} \Rightarrow State_1 \models (SA_2, RA, AA)^{Deny} \\ State_2 \models (SA_2, RA, AA)^{Permit} \Rightarrow State_2 \models (SA_2, RA, AA)^{Permit} \end{array} \right\} \Rightarrow conflict_{State_2}^{State_1}$

可以看出引起规则冲突主要有以下几种原因:

- (1) 继承自上层主体属性的正向权限与自身负向权限对下层资源属性的蕴含冲突, (ii) 属于这种情况;
(2) 继承自上层主体属性的负向权限对下层资源属性的蕴含与自身正向权限冲突, (iv) 属于这种情况;
(3) 自身负向权限对下层资源属性的蕴含与自身正向权限冲突, (vi) 属于这种情况; (4) 相同资源属性层次, 主体自身负向权限与继承自上层主体属性的正向权限冲突, (vii) 属于这种情况; (5) 相同资源属性层次, 主体自身正向权限和继承自上层主体属性的负向权限冲突, (viii) 属于这种情况. 类型 (i) (iii) (v) 不会引起规则冲突.

4.2 基于属性层次操作关联的冲突检测

利用图 1 方式可以描述 XACML 中的主体属

性层次和权限策略继承关系, 但 XACML 并没有提供专门针对资源属性层次的优化描述方案. 考虑到属性层次操作关联引起的规则冲突特点, 本节给出一种基于资源语义树的策略索引结构, 可直观表达资源属性层次关系, 迅速定位冲突相关策略, 提高冲突检测效率. 图 3 是一个已建立策略索引的资源语义树示例, 策略中定义的资源属性都可以根据其属性层次位置在语义树中找到对应节点, 树中每个 resource 节点都拥有一个策略索引列表, 存放涉及该资源属性的所有策略标识符, 标识符条目指向实际的访问控制策略, 同时将策略内部的 permit 和 deny 类型规则分类存储, 以提高分析检测效率.

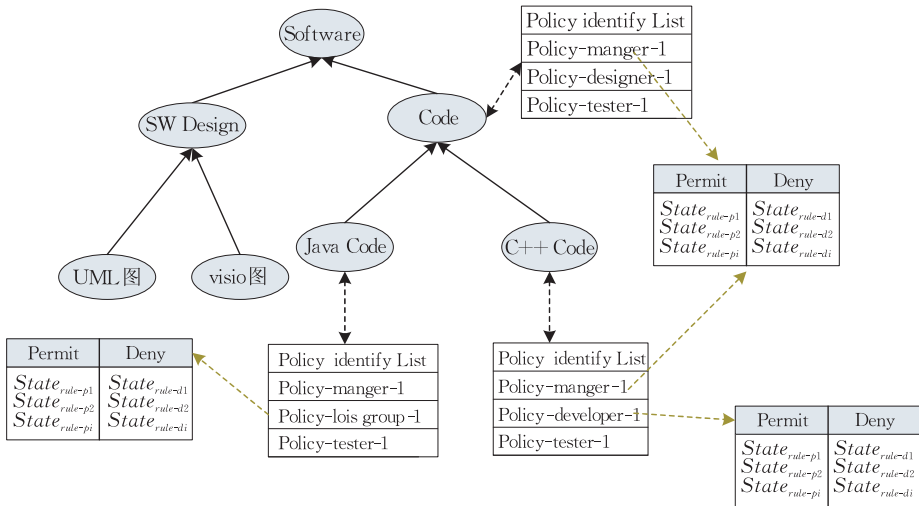


图 3 基于资源语义树的 XACML 策略索引

利用资源语义树的策略索引结构,可给出如下基于属性层次操作关联的冲突检测算法. 算法输入包括需检测的主体属性策略 pol 和其引用继承的上层主体属性的权限策略列表,输出是由若干冲突规则对 $pair(rule_p, rule_d)$ 组成的集合 $conSet$.

算法 1. 基于属性层次操作关联的冲突检测算法.

```

输入:  $pol$  //某主体属性的权限策略
       $rePoList$  //  $pol$  直接引用或间接引用的上层主体
      属性权限策略列表
输出:  $conSet$  //冲突规则对  $pair(rule_p, rule_d)$  组成的集合
begin
for each  $rule_p \in pol$ 
  for each  $resAttr_p \in rule_p.res$ 
  {
    get the  $Node_{res}$  of  $resAttr_p$ 
    for each  $pol_{\Delta} \in policyList$  of  $Node_{res}$ 
      if  $pol_{\Delta} \in rePoList$ 
        for each  $rule_d \in pol_{\Delta}$ 
          if  $State_{rule_p} \xleftrightarrow{(res, ac)} State_{rule_d}$  //第(5)种类型
            add  $pair(rule_p, rule_d)$  to  $conSet$ 
          traverse-up-checking( $Node_{res}$ )
        {
          for each  $pol_{\Delta} \in policyList$  of  $Node_{res}.parent$ 
            if  $pol_{\Delta} == pol$ 
              for each  $rule_d \in pol$ 
                if  $\exists resAttr_d \in rule_d.res; resAttr_p \triangleleft resAttr_d \wedge$ 
                   $State_{rule_p} \xleftrightarrow{(ac)} State_{rule_d}$ 
                  //第(3)种类型
                  add  $pair(rule_p, rule_d)$  to  $conSet$ 
                else if  $pol_{\Delta} \in rePoList$ 
                  for each  $rule_d \in pol_{\Delta}$ 
                    if  $\exists resAttr_d \in rule_d.res; resAttr_p \triangleleft resAttr_d \wedge$ 
                       $State_{rule_p} \xleftrightarrow{(ac)} State_{rule_d}$ 
                      //第(2)种类型
                      add  $pair(rule_p, rule_d)$  to  $conSet$ 
                    if  $Node_{res}.parent$  isn't  $root-node$ 
                      traverse-up-checking( $Node_{res}.parent$ )
                }
          }
        }
  }
for each  $rule_d \in pol$ 
  for each  $resAttr_d \in rule_d.res$ 
  {
    get the node  $Node_{res}$  of  $resAttr_d$ 
    for each  $pol_{\Delta} \in policyList$  of  $Node_{res}$ 
      if  $pol_{\Delta} \in rePoList$ 
        for each  $rule_p \in pol_{\Delta}$ 

```

```

    if  $State_{rule_p} \xleftrightarrow{(res, ac)} State_{rule_d}$  //第(4)种类型
      add  $pair(rule_p, rule_d)$  to  $conSet$ 
    traverse-down-checking( $Node_{res}$ )
  }
  for each  $node_{child} \in Node_{res}.child$ 
    for each  $pol_{\Delta} \in policyList$  of  $Node_{res}$ 
      if  $pol_{\Delta} \in rePoList$ 
        for each  $rule_p \in pol_{\Delta}$ 
          if  $\exists resAttr_p \in rule_p.res; resAttr_p \triangleleft resAttr_d \wedge$ 
             $State_{rule_p} \xleftrightarrow{(ac)} State_{rule_d}$ 
            //第(1)种类型
            add  $pair(rule_p, rule_d)$  to  $conSet$ 
          if  $node_{child}$  isn't  $leaf-node$ 
            for each  $N \in node_{child}.child$ 
              traverse-down-checking( $N$ )
        }
      }
    }
  }
return  $conSet$ 
end

```

算法具体过程如下. 针对 pol 中的 permit 类型规则 $rule_p$, 确定 $rule_p$ 中资源属性 $resAttr_p$ 在语义树上的节点 $Node_{res}$, 对 $Node_{res}$ 策略标识列表中的每个策略 pol_{Δ} 进行检测. 逐一分析 pol_{Δ} 中的 deny 类型规则 $rule_d$, 若 $rule_p$ 和 $rule_d$ 状态相关性为 $State_{rule_p} \xleftrightarrow{(res, ac)} State_{rule_d}$, 则 $pair(rule_p, rule_d)$ 属于上节分析的第(5)种类型, 将其加入 $conSet$. 调用递归函数 $Traverse-up-checking(Node_{res})$ 处理 $Node_{res}$ 上层节点的负向权限蕴含可能引发的规则冲突. $Traverse-up-checking$ 函数检索 $Node_{res}$ 父节点策略标识列表中的每个策略: 若策略 pol 包含在其中并判定 $\exists resAttr_d \in rule_d.res, rule_d \in pol; resAttr_p \triangleleft resAttr_d \wedge State_{rule_p} \xleftrightarrow{(ac)} State_{rule_d}$ 为真, 则说明策略 pol 对 $Node_{res}$ 的负向权限蕴含与自身对 $Node_{res}$ 的正向权限发生冲突, 属于第(3)种类型; 若其中存在某策略 $pol_{\Delta} \in rePoList$ 并判定 $\exists resAttr_d \in rule_d.res, rule_d \in pol_{\Delta}; resAttr_p \triangleleft resAttr_d \wedge State_{rule_p} \xleftrightarrow{(ac)} State_{rule_d}$ 为真, 则说明引用策略 pol_{Δ} 对 $Node_{res}$ 的负向权限蕴含与 pol 对 $Node_{res}$ 的正向权限发生冲突, 属于第(2)种类型. 将上述产生的规则冲突对加入 $conSet$ 中. 若 $Node_{res}$ 的父节点 $Node_{res}.parent$ 不是语义树的根节点, 则继续调用 $Traverse-up-checking(Node_{res}.parent)$. 针对 pol 中 deny 类型规则 $rule_d$,

类似第(5)种类型冲突的检测方法可以发现第(4)种类型冲突. $rule_d$ 对下层节点的负向权限蕴含与引用策略对下层节点的正向权限可能引发第(1)种类型冲突, 需要调用递归函数 $Traverse-down-checking$ ($Node_{res}$) 处理此类冲突. 其具体过程类似 $Traverse-up-checking$ 函数, 目的是遍历 $Node_{res}$ 及其子节点, 检测策略标识列表中的引用策略是否存在 $rule_p$, 使 $\exists resAttr_p \in rule_p.res; resAttr_p \triangleleft resAttr_d \wedge State_{rule_p} \xleftrightarrow{(ac)} State_{rule_d}$ 判定为真. 若存在, 将 $pair(rule_p, rule_d)$ 加入 $conSet$. 至此, 算法对上文分析的 5 种冲突类型检测完毕, 返回 $conSet$.

算法涉及的主要操作包括策略标识列表遍历和规则匹配. 设定系统策略库集合为 M , pol 中 permit 和 deny 规则定义的资源节点分别用 $\sum_{rule_p \in pol} node_{reAttr}$ 和 $\sum_{rule_d \in pol} node_{reAttr}$ 表示. $Traverse-up-checking(node_{reAttr})$ 遍历的节点表示为 $Node_{reAttr}^{up}$, $rePoList$ 中包含的 deny 规则表示为 $\sum_{rule_d \in pol} rule_d$, 策略遍历操作时间复杂性为 $\Theta(|Node_{reAttr}^{up}| \times |M|)$, 规则匹配操作的时间复杂性为 $\Theta(\sum_{rule_p \in pol} |Node_{reAttr}^{up}| \times |\sum_{rule_d \in pol} rule_d|)$; $Traverse-down-checking(node_{reAttr})$ 遍历的节点表示为 $Node_{reAttr}^{down}$, $rePoList$ 中包含的 permit 规则表示为 $\sum_{rule_p \in pol} rule_p$, 策略遍历操作时间复杂性为 $\Theta(|Node_{reAttr}^{down}| \times |M|)$, 规则匹配操作的时间复杂性为 $\Theta(\sum_{rule_d \in pol} |Node_{reAttr}^{down}| \times |\sum_{rule_p \in pol} rule_p|)$.

若系统不采用基于资源语义树的策略索引, 只使用一般的顺序列表存储策略, 则冲突检测需要遍历所有规则中的资源属性节点, 其数目用 $|\sum_{rule_i \in pol_j, pol_j \in M} node_{reAttr}|$ 表示, 冲突检测操作的时间复杂性为 $\Theta((\sum_{rule_p \in pol} node_{reAttr} + \sum_{rule_d \in pol} node_{reAttr}) \times |M| \times |\sum_{rule_i \in pol_j, pol_j \in M} node_{reAttr}|)$. 因为 $|\sum_{rule_i \in pol_j, pol_j \in M} node_{reAttr}|$ 大于 $(\sum_{rule_p \in pol} |Node_{reAttr}^{up}| \times |\sum_{rule_d \in pol} rule_d|) + (\sum_{rule_d \in pol} |Node_{reAttr}^{down}| \times |\sum_{rule_p \in pol} rule_p|)$, 且 $|\sum_{rule_i \in pol_j, pol_j \in M} node_{reAttr}| \times |M|$ 大于 $(|Node_{reAttr}^{up}| + |Node_{reAttr}^{down}|) \times |M|$, 所以基于资源语义树的策略索引结构可明显提高规则冲突的检测效率.

假设主体角色属性层次关系: $manager \triangleleft developer \triangleleft employee$, 利用算法分别检测 $(pol_{manager}, rePoList_{employee})$ 和 $(pol_{manager}, rePoList_{developer})$, $reP-$

$oList_{developer}$ 显然包含了 $rePoList_{employee}$ 中的权限策略. 由此可以看出, 高层继承检测涉及的规则匹配包含了低层继承检测涉及的规则匹配, 前者的检测过程可以在后者的计算结果基础上运行, 这个性质有助于提高系统实际的检测效率.

4.3 基于状态相关性的其他类型冲突检测

除了上节分析的规则冲突类型外, 不存在继承关系的任意策略间由于规则状态性也会导致冲突. 例如 $State_i(group.external, read, docs, time[9am, 2pm]) \mapsto permit$ 和 $State_j(role.developer \vee group.external, read \vee write, code \vee docs) \mapsto deny$ 存在 $State_i < State_j$, 同时满足两个状态空间的访问请求会导致规则冲突, 状态覆盖更详细的讨论会在第 5 节中介绍. 再如 $State_i(role.developer \vee group.external, type.code \vee type.docs, read \vee write, time[9am, 2pm]) \mapsto deny$ 和 $State_j(role.developer \vee role.testers, type.code \vee type.testsript, read \vee write, time[11am, 5pm]) \mapsto permit$ 存在

$$State_i \xleftrightarrow{(sub, role, res, type, ac, en, time)} State_j,$$

请求 $req(role.developer, type.code, read)$ 在时间段 $[11am, 2pm]$ 同时满足两个状态空间会导致判定冲突. 另外由于负向权限对下层资源权限蕴含, 上层属性节点的规则和下层节点规则如果存在状态相交也可能导致冲突. 下面是基于状态相关的其他类型冲突检测算法, 算法输入为要检测的资源属性节点 $node$, 输出是冲突规则对集合 $conSet$ 和冲突威胁规则对集合 $p-conSet$.

算法 2. 基于状态相关的其他类型冲突检测算法.

输入: $node$ // 要检测的资源属性节点

输出: $conSet$ // 冲突规则对 $pair(rule_p, rule_d)$ 组成的集合

$p-conSet$ // 冲突威胁规则对 $pair(rule_p, rule_d)$ 组成的集合

begin

get $policyList$ of $node$

for each $pol \in policyList$

{

for each $rule_p \in pol$

if $\exists resAttr \in rule_p.res; resAttr \mapsto node$
traverse-up-checking($node$)

{

get $policyList_{\Delta}$ of $node$

for each $pol_{\Delta} \in policyList_{\Delta}$

for each $rule_d \in pol_{\Delta}$

if $State_{rule_p} < State_{rule_d} \vee State_{rule_d} < State_{rule_p}$

```


$$State_{rule_p} \xleftrightarrow{(sub \vee sub.attr, res \vee res.attr, ac, en \vee en.attr)} State_{rule_d}$$

add pair( $rule_p, rule_d$ ) to conSet
else if  $State_{rule_p} \xleftrightarrow{(res \vee res.attr, ac, en \vee en.attr)} State_{rule_d}$ 
    add pair( $rule_p, rule_d$ ) to p-conSet
if node isn't root-node
    traverse-up-checking(node, parent)
}
return conSet and p-conSet
end

```

算法具体过程如下。获取 $node$ 的策略标识列表并检测其中每个策略包含的 permit 类型规则 $rule_p$ 。若 $rule_p$ 中存在资源属性对应节点 $node$ ，运行递归函数 $Traverse-up-checking(node)$ 。函数将遍历 $node$ 的祖先节点，目的是检测节点对应的策略标识列表，针对其中任意策略 pol_{Δ} 当 $rule_d \in pol_{\Delta}$ 且 $State_{rule_p} < State_{rule_d} \vee State_{rule_d} < State_{rule_p} \vee$

$State_{rule_p} \xleftrightarrow{(sub \vee sub.attr, res \vee res.attr, ac, en \vee en.attr)} State_{rule_d}$ 为真时， $rule_p$ 和 $rule_d$ 的状态向量空间存在属性分量的交集，将 $pair(rule_p, rule_d)$ 加入 $conSet$ 。

若 $State_{rule_p} \xleftrightarrow{(res \vee res.attr, ac, en \vee en.attr)} State_{rule_d}$ 为真，将 $pair(rule_p, rule_d)$ 加入 $p-conSet$ 。此类冲突是否发生依赖访问上下文信息，虽然规则的主体属性不存在交集，但用户实际访问时拥有的主体属性可能同时满足两个规则从而引起访问权限冲突，因此将 $pair(rule_p, rule_d)$ 定义为冲突威胁规则对。设节点 $node$ 的策略标识列表中 permit 规则表示为 $\sum_{rule_p \in pol}^{pol \in policyList_{node}} rule_p$ ， $Traverse-up-checking(node)$ 遍历过的节点集合表示为 $ancestorSet_{node}$ ，算法运行中需要进行规则匹配的 $deny$ 规则表示为

$\sum_{rule_d \in pol, pol \in policyList_{tNode}}^{tNode \in ancestorSet_{node}} rule_d$ ，该算法计算复杂性为 $O(|\sum_{rule_d \in pol, pol \in policyList_{tNode}}^{tNode \in ancestorSet_{node}} rule_d| \times |\sum_{rule_p \in pol}^{pol \in policyList_{node}} rule_p|)$ 。

5 规则冗余分析

根据前面状态覆盖的定义可以看出，如果 $Rule_i$ 和 $Rule_j$ 存在 $State_{Rule_i} < State_{Rule_j}$ ，那么在特定的评估结果合并算法下， $Rule_i$ 可能对访问请求的判定不产生实际影响，称这类规则为冗余规则。若冗余规则发生在策略内部，则根据规则合并算法进行冗余判定；若冗余规则发生策略间，则根据规则合并算法和策略合并算法进行冗余判定。

对图 4 中的策略集 PS 进行策略内规则冗余分析和策略间规则冗余分析。策略 P1 中 $State_{R2} < State_{R3}$ ，又因为 R3 的 $effect$ 类型为 permit 且 P1 采用允许优先合并算法，所以可推出规则 R2 冗余。需要注意的是，虽然 R1 和 R3 之间也存在关系 $State_{R1} < State_{R3}$ ，但因为正向权限不会沿资源属性路径向下层资源蕴含，所以规则 R3 针对访问请求 (admin-root/doc, write) 并不能给出明确的判定结果。同理可推出策略 P2 中规则 R5 冗余。策略 P3 中 R6 和 R7 存在关系 $State_{R7} < State_{R6}$ ，并且 R6 的 $effect$ 类型为 deny，可以沿资源属性路径向下层蕴含，其针对访问请求 (project-root/code, write) 给出判定结果 deny，所以可推出 R7 是冗余规则。完成策略内规则冗余分析后，需要进一步对策略间规则冗余进行分析。对于已经在策略内部被判定冗余的策略不再进行策略间的规则比较，例如比较策略 P1 和 P2 内的规则，只需要分别对 (R1, R4) 和 (R3, R4) 进行比较。可以看出因为 $State_{R6} < State_{R4}$ 且策略集 PS 采用首次适用策略合并算法，所以推出 R6 是冗余规则。整个规则冗余分析结果如表 1 所示。

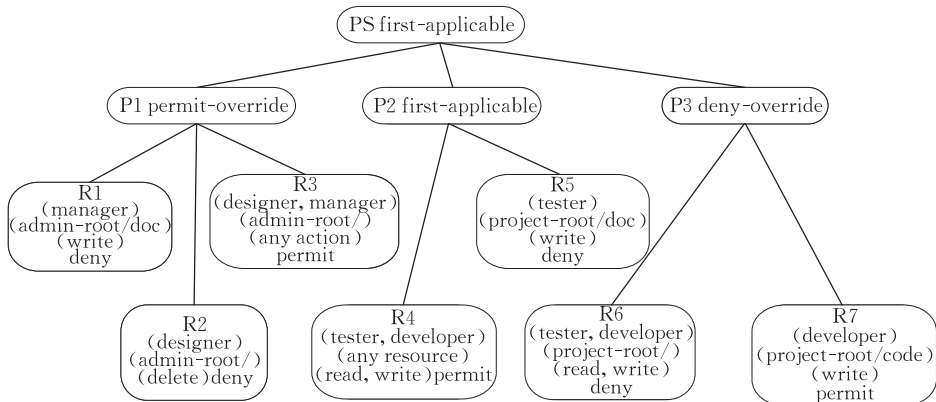


图 4 策略集 PS 结构

表 1 规则冗余分析过程及结果

策略内规则冗余分析		策略间规则冗余分析	
P1 permit-override	$(R1, R2): State_{R1} \xleftrightarrow{(sub)} State_{R2}$	(P1, P2) first-applicable	$(R1, R4): State_{R1} \xleftrightarrow{(sub, res, ac)} State_{R4}$
	$(R1, R3): State_{R1} < State_{R3}$		$(R3, R4): State_{R3} \xleftrightarrow{(res, ac)} State_{R4}$
	$(R2, R3): State_{R2} < State_{R3}, R2$ 冗余		
P2 first-applicable	$(R4, R5): State_{R5} < State_{R4}, R5$ 冗余	(P1, P3) first-applicable	$(R1, R6): State_{R1} \xleftrightarrow{(sub, ac)} State_{R6}$
			$(R3, R6): State_{R3} \xleftrightarrow{(ac)} State_{R6}$
P3 deny-override	$(R6, R7): State_{R7} < State_{R6}, R7$ 冗余	(P2, P3) first-applicable	$(R4, R6): State_{R6} < State_{R4}, R6$ 冗余

不同的规则合并算法推导出的冗余规则可能不同,根据 XACML 中提供的 permit-override、deny-override 和 first-applicable 合并算法,存在下面的定理.

定理 1. permit-override 算法下, $Rule_i$ 的 $effect$ 为任意类型, $Rule_j$ 的 $effect$ 为 permit, 存在 $State_{Rule_i} < State_{Rule_j}$, 且不存在 $res_i \triangleleft res_j$, 则 $Rule_i$ 是冗余规则.

证明. 任意访问请求 $\forall req(sub, res, ac, en)$, 若 $Rule_i \models req(sub, res, ac, en)$, 因为 $State_{Rule_i} < State_{Rule_j}$ 且不存在 $res_i \triangleleft res_j$, 可推得 $Rule_j \models req(sub, res, ac, en)$, 从而推得 $Rule_j \models req(sub, res, ac, en) \mapsto permit$. 使用 permit-override 算法合并判定结果, 得出 $combine(Rule_i, Rule_j) \mapsto permit$. 消除规则 $Rule_i$ 不影响对任意访问请求的判定结果, 因此 $Rule_i$ 是冗余规则. 若存在 $res_i \triangleleft res_j$ 且 $Rule_i \models req(sub, res_i, ac, en)$, 因为 permit 类型权限不能沿资源属性路径向下蕴含, 从 $Rule_j \models req(sub, res_j, ac, en)$ 无法推得 $Rule_j \models req(sub, res_i, ac, en) \mapsto permit$, $Rule_i$ 不是冗余规则. 证毕.

定理 2. deny-override 算法下, $Rule_i$ 的 $effect$ 为任意类型, $Rule_j$ 的 $effect$ 为 deny, 存在 $State_{Rule_i} < State_{Rule_j}$, 则 $Rule_i$ 是冗余规则.

证明. 任意访问请求 $\forall req(sub, res, ac, en)$, 若 $Rule_i \models req(sub, res, ac)$ 且不存在 $res_i \triangleleft res_j$ 的情况, 因为 $State_{Rule_i} < State_{Rule_j}$, 所以 $Rule_j \models req(sub, res, ac, en)$, 从而 $Rule_j \models req(sub, res, ac, en) \mapsto deny$. 使用 deny-override 算法合并判定结果, 得出 $combine(Rule_i, Rule_j) \mapsto deny$. 消除规则 $Rule_i$ 不影响对任意访问请求的判定结果, 因此 $Rule_i$ 是冗余规则. 若存在 $res_i \triangleleft res_j$ 且 $Rule_i \models req(sub, res_i, ac, en)$, 因为 deny 类型权限可以沿资源属性路径向下隐性传播, 从 $Rule_j \models req(sub, res_j, ac, en)$ 可以推得 $Rule_j \models req(sub, res_i, ac, en) \mapsto deny$. 对请求 $req(sub,$

$res_i, ac, en)$ 应用 deny-override 算法合并判定结果, 得出 $combine(Rule_i, Rule_j) \mapsto deny$. 消除规则 $Rule_i$ 不影响对任意访问请求的判定结果, 因此 $Rule_i$ 是冗余规则. 证毕.

定理 3. first-applicable 算法下, $seq(Rule_j) < seq(Rule_i)$, 存在 $State_{Rule_i} < State_{Rule_j}$. 若 $Rule_j.effect = deny$, 则 $Rule_i$ 是冗余规则; 若 $Rule_j.effect = permit$ 且不存在 $res_i \triangleleft res_j$, 则 $Rule_i$ 是冗余规则.

证明. 任意访问请求 $\forall req(sub, res, ac, en)$, 当 $Rule_i \models req(sub, res, ac, en)$ 时, 若 $Rule_j.effect = deny$, 因为定理 2 并且 $seq(Rule_j) < seq(Rule_i)$, 使用 first-applicable 算法合并判定结果, 得出 $combine(Rule_i, Rule_j) \mapsto Rule_j.effect$. 消除规则 $Rule_i$ 不影响对任意访问请求的判定结果, 因此 $Rule_i$ 是冗余规则. 若 $Rule_j.effect = permit$ 且不存在 $res_i \triangleleft res_j$, 因为定理 1 并且 $seq(Rule_j) < seq(Rule_i)$, 使用 first-applicable 算法合并判定结果, 得出 $combine(Rule_i, Rule_j) \mapsto Rule_j.effect$. 消除规则 $Rule_i$ 不影响对任意访问请求的判定结果, 因此 $Rule_i$ 是冗余规则. 证毕.

6 测试结果及分析

本节首先通过仿真实验分析 4.1 节规则冲突检测算法的效率和可用性. 然后利用仿真测试分析资源语义树策略索引结构和规则冗余分析技术对判定评估的性能提升.

规则冲突检测算法相关的测试主要包含 3 个实验. 图 5(a)所示的实验是分析资源语义树节点数对检测算法的效率影响. 分别构建由 20、40、60、80、100 个节点组成的资源语义树, 每棵树加载的策略总量相等(都是 200 条策略)且策略在树中节点均匀分布(根据策略内资源属性, 一条策略可以在不同节

点重复加载),5 棵树中每个节点的策略标识列表分别加载了 12、10、8、6、4 条策略. 每组策略用例中主体属性层次最高值都为 6,例如第 1 组策略中存在主体角色属性层次关系: $director \triangleleft project\ manager \triangleleft group\ leader \triangleleft senior\ employee \triangleleft junior\ employee \triangleleft intern\ student$. 冲突检测算法比较策略 pol 和引用策略列表 $rePoList$ 中的规则冲突,因此 $rePoList$ 中包含的规则总数直接影响检测时间,为了具有可比性,5 组用例中每个主体属性对应的权限策略中自包含的规则数相同(都为 10 条). 另外,主体属性所在层次不同决定了其引用的策略规则数目会有差别,例如第 1 组策略中 $junior\ employee$ 和 $director$ 引用的规则总数分别为 10 条和 50 条. 表 2 以第 1 组策略为例,描述了各层继承引用规则数目. 对各组策略内存在 1 层~5 层继承关系的属性对分别进行检测,比较不同资源语义树下的每层继承检测所花费的平均时间. 可以看出,随着节点数的增加各层继承检测时间都有所增加,原因在于检测算法需要遍历相关资源属性节点的祖先节点和后代节点,节点数增加导致算法平均遍历的节点数目也增多. 虽然语义树中节点越少其附带策略列表的长度越长,但实验数据表明,在策略加载总量足够多的情况下(本实验的策略总量是最大节点数的 2 倍),节点数增加导致的遍历计算开销要大于策略列表增长带来的计算开销. 图 5(b)的实验是在相同资源语义树下,对各组策略内存在 1 层~5 层继承关系的属性对分别进行检测,分析策略规模增长对检测效率的影响. 语义树有 50 个节点,5 组测试用例分别包含 100、200、300、400、500 条策略,每个节点的策略列表分别加载 5、10、15、20、25 条策略. 每组策略用例中主体属性层次最高值都为 6,组内每个主体属性权限策略

中的规则数分别为 10、10、20、20、30. 可以看出,随着策略规模增加和主体属性的权限规则数目增加,检测时间会显著延长. 第 1 组和第 2 组、第 3 组和第 4 组的权限规则数目相同,但第 2 组和第 4 组的策略总量较多,资源属性节点的策略标识列表长度增加,结果显示其各层继承检测的耗时都稍多一些. 策略规模增长和检测层次增高的综合作用导致:低层继承检测时,算法运行耗时虽然会随层次的增加而延长,但曲线陡增程度不是很高;高层继承检测时,曲线陡增程度明显增强且检测层次越高越明显. 高层继承检测包含低层继承检测的规则匹配,由此造成单次检测的计算资源浪费,检测系统可以先期运行低层继承检测并保存检测结果,后期运行的高层继承检测针对重复的规则匹配直接引用前期数据,通过共用相关检测结果的方法提升检测效率. 图 5(c)实验在图 5(b)实验的基础上加入了检测结果缓存,缓存中按继承层数由低到高分别存储不同(pol , $rePoList$)的检测结果. 高层继承检测时,对于已经运行过的低层继承检测运算,直接在缓存中获取检测结果. 例如对 5 层继承关系($director$, $project\ manager$)进行检测时,首先从缓存中取出 4 层继承关系($director$, $group\ leader$)的检测结果,然后只需对 $director$ 的非继承权限规则和 $project\ manager$ 的非继承权限规则进行比较检测. 图中数据表明,通过加入检测结果缓存,继承层数增多导致的检测时间差异得到有效抑制. 从 2 层继承检测开始,各层检测所花费时间的差别不大,与图 5(b)相比,不但检测速度大幅提高,而且系统表现出很好的稳定性. 以 5 层继承检测为例,各组策略用例的检测速度分别提高了 144%、148%、171%、176%和 182%.

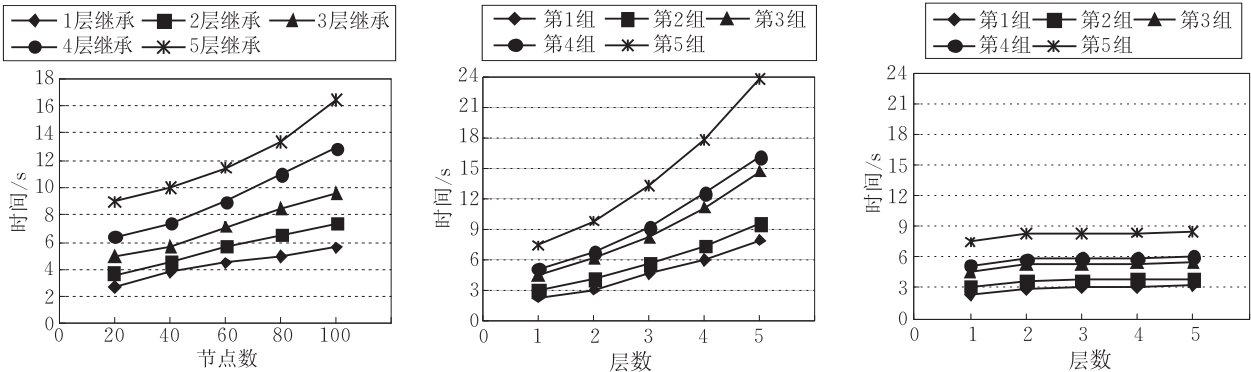


图 5 冲突规则检测算法性能实验结果

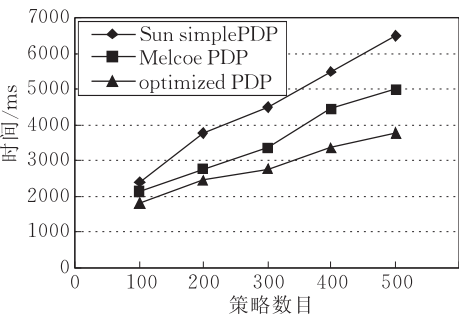
表 2 各层继承引用说明

继承关系	属性对	$(\sum rule_{pol} * \sum rule_{rePolList})$
1 层继承	<i>(junior employee, intern student) (senior employee, intern student)</i> <i>(group leader, intern student) (project manager, intern student)</i> <i>(director, intern student)</i>	(10,10)
2 层继承	<i>(senior employee, junior employee) (group leader, junior employee)</i> <i>(project manager, junior employee) (director, junior employee)</i>	(10,20)
3 层继承	<i>(group leader, senior employee) (project manager, senior employee)</i> <i>(director, senior employee)</i>	(10,30)
4 层继承	<i>(project manager, group leader) (director, group leader)</i>	(10,40)
5 层继承	<i>(director, project manager)</i>	(10,50)

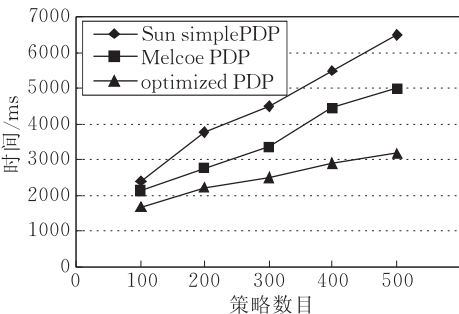
针对语义树策略索引结构和规则冗余分析提升判定性能的测试,分别提供由 100、200、300、400、500 条策略组成的 5 组策略用例,每条策略平均包含 4 条规则,每组策略内分别有 100、200、300、400、500 条冗余规则.选用三种判定系统进行分析比较,分别是:基于 Sun XACML 构建一个 simple PDP,基本没有对策略索引进行技术优化,采用列表结构顺序存储策略,也没有考虑规则冗余分析;Melcoe PDP,通过 XML 类型数据库处理 XACML 策略加快匹配速度,利用属性列表缩减策略检索空间,但没有规则冗余方面的优化;optimized PDP,采用资源语义树策略索引结构并利用规则冗余分析技术缩减策略规模. optimized PDP 是在 Sun XACML 基础上扩展实现的,其在系统初始化阶段建立策略缓存,缓存的逻辑结构是由策略中资源属性节点组成的树结构,每个节点都附带策略列表,策略在缓存中的加载过程就是根据其内部资源属性分别将策略标识添加到对应策略列表的过程. 另外,在 optimized PDP 实时运行前,可预先对策略库中所有策略条目进行冗余分析,删除冗余规则,将优化后的策略库载入策略缓存.

图 6(a)实验检测资源语义树策略索引结构对判定性能的提升,分别构造 100、200、300、400、500 条访问请求应用于 5 组策略用例,计算平均每条请求完成访问判定的时间.当基于较小规模策略时,

simple PDP 和其他两种系统的判定时间相差不大,但随着策略规模成倍增加,其判定性能差距显著表现出来.由于没有进行任何策略索引方面的优化, simple PDP 面对规模较大的策略用例仍然采用逐条策略完全匹配的方式进行判定.假设分别对 N 个资源发出访问请求,策略库规模为 M , simple PDP 的策略匹配操作的计算复杂性为 $\Theta(N \times M)$. optimized PDP 由于采用资源语义树索引结构,其策略匹配的计算复杂性为 $\Theta(\sum_{i=1}^N policyList_i.length)$, $policyList_i.length$ 在最坏情况下取值为 M , $\Theta(N \times M)$ 是 optimized PDP 计算复杂性的极限最大值,而且在实际系统中这种情况基本不可能出现,因此 optimized PDP 的判定性能优势会随着策略规模增长表现得更加显著,图中的曲线走势也验证了这一推论.图 6(b)实验中的 optimized PDP 在系统运行前加入冗余分析过程,实现了对策略库规模的优化,其判定性能相比(a)得到进一步的提升.可以看出, Melcoe PDP 总体评估性能还是不错的,其通过属性列表缩减策略检索空间并采用 XML 数据库加速策略解析速度,但这种做法牺牲了 XACML 策略一定的细粒度控制能力,系统潜在的复杂安全需求可能导致管理员频繁改动属性列表. optimized PDP 相比 Melcoe PDP,效率优势在策略规模较大情况下表现得更加明显.



(a) 未加冗余



(b) 加入冗余

图 6 访问判定性能比较

以上仿真测试的实验环境为: Intel Pentium4 2.4GHz CPU, 1GB 内存, Windows XP SP2 操作系统平台, Java Runtime Environment1.5.08.

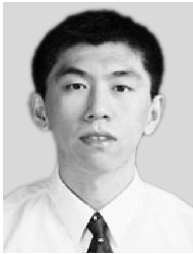
7 结束语

Policy-based management 是新一代互联网体系结构下大规模资源访问管理的发展趋势. 本文分析了 XACML 中属性层次操作关联引发的各种规则冲突类型, 基于资源语义树建立策略索引结构, 并在此基础上分别给出基于属性层次操作关联的冲突检测算法和基于状态相关性的其他类型冲突检测算法. 利用状态覆盖的思想推导了多种合并算法下的规则冗余判定定理. 最后通过仿真实验分析了多种情况下冲突检测算法的运行效率, 验证语义树策略索引和冗余规则处理可有效改进判定性能.

今后的工作主要集中在: 冲突检测算法的具体优化实现, 进一步提升检测性能; 根据不同的冲突类型, 扩展自定义的冲突合并消解算法; 提升 optimized PDP 策略规则解析匹配速度, 使其更具实际工程意义.

参 考 文 献

- [1] Sloman M. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 1994, 2(4): 333-360
- [2] Moses T. eXtensible access control markup language (XACML) version 2.0. OASIS Standard, 2005
- [3] Jajodia S, Samarati P, Subrahmanian V S et al. A unified framework for enforcing multiple access control policies// *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Tucson, Arizona, USA, 1997, 26(2): 474-485
- [4] Jajodia S, Samarati P, Subrahmanian V S. A logical language for expressing authorizations// *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. Los Alamitos, California, USA, 1997: 31-42
- [5] Lupu E, Sloman M. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 1999, 25(6): 852-869
- [6] Cholvy L, Cuppens F. Analyzing consistency of security policies// *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. Los Alamitos, California, USA, 1997: 103-112
- [7] Dunlop N, Indulska J, Raymond K. Dynamic conflict detection in policy-based management systems// *Proceedings of the 6th International Enterprise Distributed Object Computing Conference (EDOC)*. Lausanne, Switzerland, 2002: 15-26
- [8] Guelev D P, Ryan M, Schobbens P Y. Model-checking access control policies. *Lecture Notes in Computer Science* 3225. Berlin: Springer-Verlag, 2004: 219-230
- [9] Zhang N, Ryan M, Guelev D P. Synthesising verified access control systems in XACML// *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*. Washington, DC, USA, 2004: 56-65
- [10] Zhang N, Ryan M, Guelev D P. Evaluating access control policies through model checking// *Proceedings of the 8th Information Security Conference (ISC)*. *Lecture Notes in Computer Science* 3650. Berlin: Springer-Verlag, 2005: 446-460
- [11] Hughes G, Bultan T. Automated verification of XACML policies using a SAT solver. *International Journal on Software Tools for Technology Transfer (STTT)*, 2008, 10(6): 503-520
- [12] Hughes G, Bultan T. Automated verification of access control policies. Computer Science Department, University of California: Technical Report 2004-22, 2004
- [13] Bryans J. Reasoning about XACML policies using CSP// *Proceedings of the 2005 Workshop on Secure Web Services (SWS)*. Fairfax, Virginia, USA, 2005: 28-35
- [14] Fisler K, Krishnamurthi S, Meyerovich L A et al. Verification and change-impact analysis of access-control policies// *Proceedings of the 27th International Conference on Software Engineering*. St. Louis, MO, USA, 2005: 196-205
- [15] Martin E, Xie T. Automated test generation for access control policies via change-impact analysis// *Proceedings of the 2nd International Workshop on Software Engineering for Secure Systems (SESS)*. Minneapolis, Minnesota, USA, 2007: 5-11
- [16] Hu V C, Martin E, Hwang J, Xie T. Conformance checking of access control policies specified in XACML// *Proceedings of the 1st IEEE International Workshop on Security in Software Engineering (IWSSE)*. Beijing, China, 2007: 275-280
- [17] Martin E, Xie T. A fault model and mutation testing of access control policies// *Proceedings of the 16th International Conference on World Wide Web (WWW)*. Banff, Alberta, Canada, 2007: 667-676
- [18] Mazzoleni P, Crispo B, Bertino E. XACML policy integration algorithms. *ACM Transactions on Information and Systems Security*, 2008, 11(1): Article 4
- [19] Kolovski V, Hendler J, Parsia B. Analyzing Web access control policies// *Proceedings of the 16th International Conference on World Wide Web*. Banff, Alberta, Canada, 2007: 677-686
- [20] Lorch M, Adams D B, Kafura D et al. The PRIMA system for privilege management, authorization and enforcement in grid environments// *Proceedings of the 4th International Workshop on Grid Computing*. Phoenix, Arizona, USA, 2003: 109-116



WANG Ya-Zhe, born in 1979, Ph. D. candidate. His research interests include information system security and distributed computing.

FENG Deng-Guo, born in 1965, professor, Ph. D. supervisor. His research interests focus on network and information security.

Background

This work is supported by the National High Technology Research and Development Program (863 Program) of China (No. 2006AA01Z454); the National Key Technology R&D Program of China (No. 2006BAH02A02) and the National Natural Science Foundation of China (No. 60603017).

One important research field of these projects is to providing policy based authorization components for distributed computing and cross domain applications. The projects apply XACML to define access control policies with respect to its flexible expressive function based on attributes, but it lacks the capabilities of detecting and pre-locating conflict rules before system makes policy decisions. While the language has proposed several combining algorithms in order to resolve conflict decisions, user doesn't know conflict origin all the time and whether the chosen algorithm is consistent with authorization intention. In addition, some rules couldn't be im-

posed on the decisions exactly under a given algorithm, these redundancy rules will lower the efficiency of policy evaluation.

This paper first introduces the concept of rule state, then designs an effective policies index architecture using resource semantic tree. After analyzing conflict categories caused by operation association between subjects' and resources' attribute hierarchy, two conflict detecting algorithms are proposed. These algorithms can pre-locate conflict rules resulting from permission inheritance and permission implication or conflict rules for the node has been specified in the semantic tree. Some rule redundancy judgment theorems for various combining algorithms are deduced. Evaluation systems that are enhanced by policies index and rule redundancy would improve match efficiency.