软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(2):323−338 [doi: 10.3724/SP.J.1001.2011.03707]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于多层次优化技术的 XACML 策略评估引擎*

王雅哲[1,2+], 冯登国[1,2], 张立武[1], 张 敏[1]

[1](中国科学院 软件研究所 信息安全国家重点实验室,北京 100190)

[2](信息安全共性技术国家工程研究中心,北京 100190)

XACML Policy Evaluation Engine Based on Multi-Level Optimization Technology

WANG Ya-Zhe[1,2+], FENG Deng-Guo[1,2], ZHANG Li-Wu[1], ZHANG Min[1]

[1](State Key Laboratory of Information Security, Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

[2](National Engineering Research Center of Information Security, Beijing 100190, China)

+ Corresponding author: E-mail: yz.wang@is.iscas.ac.cn

Wang YZ, Feng DG, Zhang LW, Zhang M. XACML policy evaluation engine based on multi-level optimization technology. *Journal of Software*, 2011,22(2):323−338. http://www.jos.org.cn/1000-9825/3707.htm

Abstract: This paper proposes an implementation scheme of XACML (extensible access control markup language) policy evaluation engine based on multi-level optimization technology, MLOBEE (multi-level optimization based evaluation engine). Before evaluating these policies, the scenario implements rule refinement to lessen scale policies and adjust the sequence at the rule. During evaluation, the engine adopts a multi-cache mechanism that includes result cache, attribute cache, and policy cache to reduce the communication cost between engine and other components. To decrease matching magnitudes and enhance matching exactitudes, policy cache practices two stage index techniques. Finally, emulation tests validate that the overall evaluation performance of MLOBEE, using multi-level optimization technology, is better than most other similar systems.

Key words: XACML (extensible access control markup language); access control; policy evaluation; rule refining; cache mechanism; policy index

摘 要: 给出一种采用多层次优化技术的 XACML(extensible access control markup language)策略评估引擎实现方案 MLOBEE(multi-level optimization based evaluation engine).策略判定评估前,对原始策略库实施规则精化,缩减策略规模并调整规则顺序;判定评估过程中,在引擎内部采用多种缓存机制,分别建立判定结果缓存、属性缓存和策略缓存,有效降低判定引擎和其他功能部件的通信损耗.通过两阶段索引实现的策略缓存,可显著降低匹配运算量并提高策略匹配准确率.仿真实验验证了 MLOBEE 所采用的多层次优化技术的有效性,其整体评估性能明显优于大多数同类系统.

关键词: XACML;访问控制;策略评估;规则精化;缓存机制;策略索引

中图法分类号: TP309 **文献标识码**: A

* 基金项目: 国家自然科学基金(61003228, 60803129); 中国科学院知识创新工程项目(YYYJ-1013); 下一代互联网业务试商用及设备产业化专项(CNGI-09-03-03)

收稿时间: 2009-02-20; 定稿时间: 2009-08-12

访问控制标记语言 XACML(extensible access control markup language)已逐渐成为多个企业应用和商业产品实现安全授权功能的实际标准[1].分布式资源共享、Web 服务、域间协作等新兴业务需要制定大量的 XACML 策略条目对资源进行细粒度访问控制,但随着策略规模和策略语义复杂性的上升,策略评估效率已成为制约系统可用性的关键瓶颈.XACML 规范[2]中虽然给出了访问控制实施框架,但没有提供策略分析、规则匹配、判定响应等相关的优化处理方法,这在很大程度上导致了 XACML 策略评估引擎在处理策略信息检索、多策略匹配等问题时的实际性能指标偏低,具体表现为系统资源开销大、访问请求应答延时长、远程通信交互多,因而无法满足商业应用的高业务吞吐量.现有 XACML 相关研究工作[3−10]主要集中在策略的建模、验证、分析和测试上,大多关注策略的有效性和正确性检验,基本上忽视了大规模策略条件下的系统响应效率问题.目前的一些判定引擎系统[11−19]虽然在不同程度上采用了部分优化技术以提高引擎判定效率,但在诸如策略规模缩减、高效策略索引等大规模策略处理关键技术上没有显著突破.

针对上述问题,本文提出一种实现多层次优化技术的策略评估引擎方案(multi-level optimization based evaluation engine,简称 MLOBEE).在策略判定之前,MLOBEE 对原始策略库进行规则精化处理,缩减策略规模;在策略评估过程中分别建立判定结果缓存、属性缓存和策略缓存,其作用在于降低评估引擎和其他功能部件的通信损耗;策略缓存内部分别以资源属性条目和权限原语为键值建立两阶段索引,提高策略匹配速度和准确率.另外,MLOBEE 采用的规则精化可以作为一种共性的策略优化技术,辅助提升其他引擎的运行效率.本文从策略加载方式、策略匹配模式以及专有优化技术原理等方面对各评估引擎进行分析;通过仿真测试比较,验证了 MLOBEE 多层次优化技术的有效性及其整体评估性能的效率优势.

1 相关工作

在 OASIS 组织制定 XACML 标准后不久,Sun 公司即推出其策略评估系统原型 Sun XACML[11],它通过策略检索模块接口和属性检索模块接口实现对访问请求的判定评估,用户可以自定义模块接口的具体实现.其后出现的多种访问判定系统大多采用了这种模式或该模式的改进方案.虽然该引擎给出了一个比较完整的访问控制功能体系结构,但其没有进一步考虑策略检索和属性检索对评估引擎带来的效率影响,只采用穷举遍历策略库的方式处理访问请求.另有一些简单的实验型判定引擎(例如 XACML.NET[12],Parthenon XACML[13])仅支持简单的策略逐条匹配方式,在策略检索和辅助功能部件方面缺乏实用性和延展性.在系统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,穷举匹配的模式会严重降低系统检索有效策略的概率.

有一些系统为了应对企业级的策略处理规模,采用了部分有针对性的引擎优化技术.JBoss XACML[14]在 Sun XACML 的基础上实现了一次上层封装,提供灵活的策略加载方式和引擎启动方式,但没有改善策略检索匹配模式以解决评估效率问题.Melcoe PDP[15]针对 XACML 策略载体的特点,将策略文件存储在 XML Native 数据库中,利用 Native 数据库对 XML 的专有处理技术提高判定引擎的策略匹配速度,并使用属性约束列表进一步缩减策略检索空间,但其优化方案依赖专有数据库自身的技术特点.XACMLight[16]提供了 Web 服务模式的策略判定和策略管理.它专注于评估引擎的远程服务调用,没有对策略匹配过程作特别的优化.AXESCON XACML[17]强化评估引擎中的策略载入和策略缓存功能,考虑了策略引用和多策略匹配问题,但其采用的匹配逻辑仍然按照 XACML 嵌套结构逐层进行,没有在匹配逻辑优化和高效索引结构方面进一步提高.

在策略索引方面比较有成效的工作是 Enterprise XACML[18]系统.它根据策略目标中包含的属性标记建立策略索引结构,在一定程度上缩减了策略检索空间,但策略目标和访问请求中可能包含多个属性标记,从而造成策略重复索引和多次匹配;另外,其索引结构没有考虑规则目标的匹配优化问题.XEngine 系统[19]采用的优化思想不同于大多数判定引擎.该方案将 XACML 策略规则转译为数字区间规则,把嵌套递归式描述结构转化为扁平结构,采用“首次适用”合并算法避免规则的遍历匹配.虽然数字区间匹配的效率高于策略字符串匹配,但引擎需要动态生成大量的辅助运算数据结构,其额外的规则转译运算量不能忽视,另外,数字区间规则并不能完全支

持 XACML 的复杂描述能力.

在评估引擎性能比较方面,文献[20]集中测试了 Sun XACML,XACML.NET 和 Parthenon XACML,在输入相同策略集和访问请求的情况下,比较输出判定结果之间的差异,从而检测引擎的内部功能缺陷.该测试工作主要研究引擎功能正确性检验问题,不考虑策略判定性能.文献[21]系统地比较了各种引擎的评估效率,分别在多策略组合和多规则组合场景下对 Sun XACML,XACMLight 和 Enterprise XACML 进行了仿真评测,探讨了性能差异原因.该测试工作验证了策略索引优化对判定性能的提升,从策略加载和策略评估两个阶段分析优化实施技术.

可以看出,大多数现有策略评估引擎所采用的优化方案提升评估效率的能力有限,其原因在于,XACML 复杂的嵌套递归结构增加了优化工作的难度;性能测试涵盖的引擎系统不够丰富,缺乏从技术原理角度对各引擎的专有技术差异的比较分析.

国内方面的工作大多集中在 XACML 策略分析及策略描述功能扩展上,缺乏评估引擎效率优化相关的科研进展.文献[22,23]主要针对 XACML 策略管理进行研究,提出了一种管理策略优化处理方案并扩展 XACML 模式以支持委托授权;文献[24]针对可信计算平台上的访问控制问题,设计了一种基于平台可信度的 XACML 策略合成算法,扩展评估引擎的策略匹配逻辑从而满足可信评估需求;本文的前期工作[25]重点分析了 XACML 策略中的规则冲突和规则冗余,并给出具体的冲突检测算法.

2 预备知识

XACML 是一种基于属性访问控制模型的策略描述语言及协议栈.它采用树状分层嵌套结构定义访问权限,如图 1 所示.

定义 1(策略容器(collection)). 策略容器分为 *PolicySet* 和 *Policy* 两种,都是独立的嵌套式结构,可以作为策略树的根节点,*PolicySet* 包含若干 *Policy* 和其他 *PolicySet*,*Policy* 由若干规则 *Rule* 组成.

定义 2(规则(rule)). 规则是策略树的叶子节点,也是最小单位的策略原语,由目标元素 *Target* 和 *effect* 元素组成,*Target* 限定规则的具体权限,*effect* 根据其自身取值表示该规则是肯定授权(*effect* 取值为 *permit*)或否定授权(*effect* 取值为 *deny*).

定义 3(属性(attribute)). XACML 中的属性类型包括 4 种:主体属性 *subAttr* 限定访问主体相关信息,资源属性 *resAttr* 限定请求资源相关信息,动作属性 *acAttr* 限定访问动作类型,环境属性 *enAttr* 限定访问上下文信息(时间、位置等).

定义 4(目标(target)). 目标元素 *Target* 由主体属性集合(*Target.sub*)、资源属性集合(*Target.res*)、动作属性集合(*Target.ac*)以及环境属性集合(*Target.en*)组成,表示“在满足环境属性约束的情况下,允许或拒绝主体属性持有者对资源属性限定的资源实施动作属性指示的具体操作”,其包含的权限范围由 4 类属性子集笛卡尔积运算结果组成,记为(*Target.sub*)⊗(*Target.res*)⊗(*Target.ac*)⊗(*Target.en*)(为简化描述,下文不再考虑环境属性).

PolicySet 和 *Policy* 也有目标元素,用来约束各自的适用范围,XACML 标准推荐了两种策略容器目标元素的计算方法:方法 1,顶层 *PolicySet* 或 *Policy* 的 *Target* 通过计算其包含的下层 *PolicySet*,*Policy* 和 *Rule* 元素各自 *Target* 的并集得到;方法 2,顶层组件的 *Target* 可以通过计算其包含所有下层组件 *Target* 的交集得到.两种方法的评估逻辑是有差别的:第 1 种情况,顶层组件的目标元素适用所有满足至少 1 个下层组件目标元素的访问请求;第 2 种情况,顶层组件的目标元素仅仅适用于满足所有下层组件目标元素的访问请求.本文中出现的策略容器目标都采用方法 1 的并集计算,例如,图 1 中描述的两条策略(策略 1 省略了主体属性和动作属性的并集计算,策略 2 省略了动作属性的并集运算).

定义 5(评估合并算法(combine algorithm)). 如果 *Policy* 内的多条 *Rule* 或者 *PolicySet* 内的多条 *Policy* 都匹配访问请求,则 XACML 利用规则/策略评估合并算法计算最终的评估结果.典型算法包括 permit-override(肯定优先)、deny-override(否定优先)、first-applicable(首次适用)和 only-one-applicable(唯一适用).

```
⟨Policy PolicyId="1" CombiningAlgId="Permit-Overrides"⟩
⟨Target⟩

 ⟨Resources⟩⟨Resource⟩ resAttr1 ⟨/Resource⟩
            ⟨Resource⟩ resAttr2 ⟨/Resource⟩
            ⟨Resource⟩ resAttr3 ⟨/Resource⟩
            ⟨Resource⟩ resAttr4 ⟨/Resource⟩
            ⟨Resource⟩ resAttr5 ⟨/Resource⟩
            ⟨Resource⟩ resAttr6 ⟨/Resource⟩⟨/Resources⟩

⟨/Target⟩
⟨Rule RuleId="1" Effect="Deny"⟩
 ⟨Target⟩
  ⟨Subjects⟩⟨Subject⟩ subAttr1 ⟨/Subject⟩
           ⟨Subject⟩ subAttr2 ⟨/Subject⟩⟨/Subjects⟩
  ⟨Resources⟩⟨Resource⟩ resAttr1 ⟨/Resource⟩
             ⟨Resource⟩ resAttr2 ⟨/Resource⟩⟨/Resources⟩
  ⟨Actions⟩⟨Action⟩ acAttr1 ⟨/Action⟩
           ⟨Action⟩ acAttr2 ⟨/Action⟩⟨/Actions⟩
 ⟨/Target⟩⟨/Rule⟩
⟨Rule RuleId="2" Effect="Permit"⟩
 ⟨Target⟩
  ⟨Subjects⟩⟨Subject⟩ subAttr2 ⟨/Subject⟩
           ⟨Subject⟩ subAttr3 ⟨/Subject⟩⟨/Subjects⟩
  ⟨Resources⟩⟨Resource⟩ resAttr3 ⟨/Resource⟩⟨/Resources⟩
  ⟨Actions⟩⟨Action⟩ acAttr3 ⟨/Action⟩⟨/Actions⟩
 ⟨/Target⟩⟨/Rule⟩
⟨Rule RuleId="3" Effect="Deny"⟩
 ⟨Target⟩
  ⟨Subjects⟩⟨Subject⟩ subAttr3 ⟨/Subject⟩
           ⟨Subject⟩ subAttr4 ⟨/Subject⟩⟨/Subjects⟩
  ⟨Resources⟩⟨Resource⟩ resAttr3 ⟨/Resource⟩
             ⟨Resource⟩ resAttr5 ⟨/Resource⟩⟨/Resources⟩
  ⟨Actions⟩⟨Action⟩ acAttr4 ⟨/Action⟩⟨/Actions⟩
 ⟨/Target⟩⟨/Rule⟩
⟨Rule RuleId="4" Effect="Permit"⟩
 ⟨Target⟩
  ⟨Subjects⟩⟨Subject⟩ subAttr4 ⟨/Subject⟩⟨/Subjects⟩
  ⟨Resources⟩⟨Resource⟩ resAttr4 ⟨/Resource⟩
             ⟨Resource⟩ resAttr6 ⟨/Resource⟩⟨/Resources⟩
  ⟨Actions⟩⟨Action⟩ acAttr4 ⟨/Action⟩⟨/Actions⟩
 ⟨/Target⟩⟨/Rule⟩
⟨/Policy⟩
```

```
⟨Policy PolicyId="2" CombiningAlgId="Deny-Overrides"⟩
⟨Target⟩
 ⟨Subjects⟩⟨Subject⟩ subAttr2 ⟨/Subject⟩
           ⟨Subject⟩ subAttr4 ⟨/Subject⟩
           ⟨Subject⟩ subAttr5 ⟨/Subject⟩⟨/Subjects⟩
 ⟨Resources⟩⟨Resource⟩ resAttr1 ⟨/Resource⟩
            ⟨Resource⟩ resAttr2 ⟨/Resource⟩
            ⟨Resource⟩ resAttr3 ⟨/Resource⟩
            ⟨Resource⟩ resAttr4 ⟨/Resource⟩⟨/Resources⟩
 ⟨/Target⟩
⟨Rule RuleId="5" Effect="Permit"⟩
 ⟨Target⟩
  ⟨Subjects⟩⟨Subject⟩ subAttr2 ⟨/Subject⟩
           ⟨Subject⟩ subAttr4 ⟨/Subject⟩⟨/Subjects⟩
  ⟨Resources⟩⟨Resource⟩ resAttr1 ⟨/Resource⟩
             ⟨Resource⟩ resAttr2 ⟨/Resource⟩⟨/Resources⟩
  ⟨Actions⟩⟨Action⟩ acAttr1 ⟨/Action⟩
           ⟨Action⟩ acAttr4 ⟨/Action⟩⟨/Actions⟩
 ⟨/Target⟩⟨/Rule⟩
⟨Rule RuleId="6" Effect="Deny"⟩
 ⟨Target⟩
  ⟨Subjects⟩⟨Subject⟩ subAttr4 ⟨/Subject⟩
           ⟨Subject⟩ subAttr5 ⟨/Subject⟩⟨/Subjects⟩
  ⟨Resources⟩⟨Resource⟩ resAttr2 ⟨/Resource⟩
             ⟨Resource⟩ resAttr3 ⟨/Resource⟩⟨/Resources⟩
  ⟨Actions⟩⟨Action⟩ acAttr2 ⟨/Action⟩
           ⟨Action⟩ acAttr3 ⟨/Action⟩
           ⟨Action⟩ acAttr4 ⟨/Action⟩⟨/Actions⟩
 ⟨/Target⟩⟨/Rule⟩
⟨Rule RuleId="7" Effect="Permit"⟩
 ⟨Target⟩
  ⟨Subjects⟩⟨Subject⟩ subAttr4 ⟨/Subject⟩⟨/Subjects⟩
  ⟨Resources⟩⟨Resource⟩ resAttr2 ⟨/Resource⟩⟨/Resources⟩
  ⟨Actions⟩⟨Action⟩ acAttr1 ⟨/Action⟩
           ⟨Action⟩ acAttr3 ⟨/Action⟩⟨/Actions⟩
 ⟨/Target⟩⟨/Rule⟩
⟨Rule RuleId="8" Effect="Permit"⟩
 ⟨Target⟩
  ⟨Subjects⟩⟨Subject⟩ subAttr2 ⟨/Subject⟩⟨/Subjects⟩
  ⟨Resources⟩⟨Resource⟩ resAttr1 ⟨/Resource⟩⟨/Resources⟩
  ⟨Actions⟩⟨Action⟩ acAttr1 ⟨/Action⟩⟨/Actions⟩
 ⟨/Target⟩⟨/Rule⟩
⟨/Policy⟩
```

Fig.1 XACML policy example

图 1 XACML 策略示例

3 MLOBEE 系统平台

本节详细阐述基于多层次优化技术的评估引擎 MLOBEE.首先给出 MLOBEE 系统平台的体系结构和功能部件组成;然后介绍 MLOBEE 引擎中采用的核心优化技术:规则精化和多级缓存机制,尤其是两阶段策略索引技术;最后说明 MLOBEE 引擎对访问请求的完整评估过程.

3.1 MLOBEE体系结构

根据 XACML 标准中提出的数据流框架,实现一个完整的访问控制系统通常需要策略执行点、策略决策点、策略信息点、策略管理点等功能部件.据此,MLOBEE 系统除了实现策略评估相关的核心功能以外,还提供完整的访问授权支撑平台.如图 2 所示,其主要功能部件包括策略管理服务部件(policy management service,简称 PMS)、策略决策服务部件(policy decision service,简称 PDS)、策略持久化服务部件(policy persistence service,简称 PPS)和属性断言服务部件(attribute assertion service,简称 AAS).

PMS 提供一个集中式的图形化策略管理平台,其主要负责策略操作、规则冗余分析及规则精化.策略操作涉及策略的创建、修改、删除和更新等;规则冗余分析根据规则评估合并算法,检测不产生实际判定影响的冗余规则;规则精化根据冗余分析结果与合并算法类型缩减策略规模并调整规则顺序,实现策略匹配的前期优化.PMS 针对 XACML 基于属性的描述框架,支持属性层次分析以及各种访问控制模型的策略定义(基于身份访问

控制、基于格的访问控制、基于角色访问控制等).另外,管理平台通过策略持久化服务远程调用获取策略信息,并转化为自身应用的策略实体.

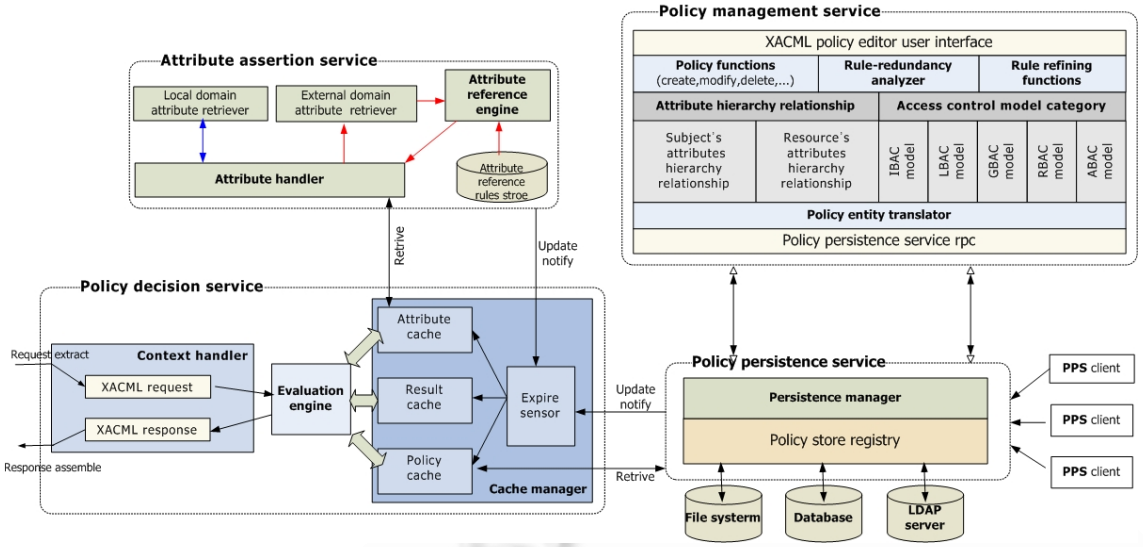


Fig.2 Architecture of MLOBEE

图 2 MLOBEE 系统架构

PDS 是 MLOBEE 系统的核心部件,多级缓存机制以及策略缓存中的两阶段策略索引都在该模块内实现.上下文处理器将具体的用户资源请求过程抽取为 XACML 请求,发送到评估引擎,将返回的判定结果装配为 XACML 应答;评估引擎利用缓存管理器中的多级缓存对当前请求进行策略匹配和访问判定,其中包含了判定结果缓存、属性缓存以及策略缓存,还提供有效期感知器监测缓存生命周期.

PPS 负责实现策略持久化存储并支持多种策略存储方式.持久化管理器负责整个持久化服务的全局系统配置,可动态加载和注销多个策略库;策略库提供者必须通过策略库注册组件登记才能被持久化管理器识别;具体的策略库提供者可以是数据库、LDAP 服务器、文件系统等.

AAS 提供属性断言发布服务,为策略匹配提供必要的属性检索功能.属性请求处理器响应来自 PDS 的属性检索请求,将查询到的属性以标准格式返回(例如 SAML 断言或属性证书);AAS 除了支持管理域内属性查询以外,还支持外域用户的属性检索;通过属性推理引擎和推理规则对域外属性进行域内属性的等价推导.

3.2 规则精化

策略库为评估引擎提供判定决策依据,规则条目数量和组织顺序决定了匹配运算规模和评估效率.数以千计的策略中可能存在若干对访问请求不产生实际判定影响的冗余规则;另外,在特定规则评估合并算法下,规则排列顺序对引擎实际的匹配运算量有直接相关性.规则精化将从以上两个方面对策略库进行优化,以辅助评估引擎提高判定效率.

假设用户发出的访问请求记为 $req(sub,res,ac)$.其中,sub,res 和 ac 分别代表访问相关的主体信息、资源信息和动作信息,其取值可以是单一属性值或是包含多个值的属性集合.若规则 R 适用于 $req(sub,res,ac)$,则记为 $R \vDash req(sub,res,ac)$.当 $R_i \vDash req(sub,res,ac)$成立时,$R_j \vDash req(sub,res,ac)$必然成立,反之则不一定,称 R_j 覆盖 R_i,记为 $R_i \lhd R_j$.此时,删除 R_i 后并不影响引擎的判定结果,称 R_i 是冗余规则.以下是由本文前期工作中的冗余判定定理[25]扩展推导出的详细冗余判定规则.

在 permit-overrides 算法下:

规则 1. 若 $R_i \lhd R_j$ 且 $R_j.effect=permit$,则 R_i 是冗余规则.

证明:对于 R_i 适用的任意访问请求 $req(sub,res,ac)$,因为 $R_i \lhd R_j$,可得 $R_j \models req(sub,res,ac)$.若 $R_i.effect=permit$,则与 $R_j.effect$ 重复,删除 R_i 不影响整个策略的最终判定结果,证得 R_i 是冗余规则;若 $R_i.effect=deny$,则因为策略采用 permit-overrides 算法,R_i 对 $req(sub,res,ac)$的判定结果被 R_j 屏蔽,删除 R_i 不影响整个策略的最终判定结果,证得 R_i 是冗余规则. □

规则 2. 若 $R_i \lhd R_j$ 且 $R_j.effect=deny$,则:当 $R_i.effect=deny$ 时,R_i 是冗余规则;当 $R_i.effect=permit$ 时,R_i 不是冗余规则.

证明:对于 R_i 适用的任意访问请求 $req(sub,res,ac)$,因为 $R_i \lhd R_j$,可得 $R_j \models req(sub,res,ac)$.若 $R_i.effect=deny$,则与 $R_j.effect$ 重复,删除 R_i 不影响整个策略的最终判定结果,证得 R_i 是冗余规则;若 $R_i.effect=permit$,则因为策略采用 permit-overrides 算法,R_i 对 $req(sub,res,ac)$的判定结果为 $permit$,R_j 无法屏蔽此结果,删除 R_i 将影响整个策略对 $req(sub,res,ac)$的最终判定结果,证得 R_i 不是冗余规则. □

在 deny-overrides 算法下:

规则 3. 若 $R_i \lhd R_j$ 且 $R_j.effect=deny$,则 R_i 是冗余规则.

规则 4. 若 $R_i \lhd R_j$ 且 $R_j.effect=permit$:当 $R_i.effect=permit$ 时,R_i 是冗余规则;当 $R_i.effect=deny$ 时,R_i 不是冗余规则.

在 first-applicable 算法下,规则 R_i 在策略中的位置顺序记为 $seq(R_i)$:

规则 5. 若 $R_i \lhd R_j$ 且 $R_i.effect=R_j.effect$,则 R_i 是冗余规则.

规则 6. 若 $R_i \lhd R_j$ 且 $R_i.effect \neq R_j.effect$,则:当 $seq(R_j)<seq(R_i)$时,R_i 是冗余规则;当 $seq(R_i)<seq(R_j)$时,R_i 不是冗余规则.

限于篇幅,规则 3~规则 6 的证明从略.

根据 only-one-applicable 算法的判断逻辑,若策略中的规则 R_i 和 R_j 同时适用某一请求,则评估引擎无法给出最终决策.因此,该算法下的任何规则冗余情况都会被系统当作错误处理,此处不再分析.另外,由于 XACML 策略容器是分层结构,因此规则冗余分析必须按照由低层容器到高层容器的顺序进行,否则会造成有效规则丢失,引发授权策略安全漏洞.

定义 6(规则精化(rule refining)). 设策略容器的规则集为 RS,根据其评估合并算法α,运行策略处理流程 $process^{\alpha}_{refine}$,获得 RS 的一个子集 $RS' \subseteq RS$.对于任何访问请求 req 来说,若 RS'和 RS 的判定评估结果保持不变,则称 RS'是 RS 在α算法下的规则求精,称过程 $process^{\alpha}_{refine}$ 为规则精化.

$process^{permit}_{refine}$ 首先根据评估合并算法对 RS:$\{R_1,R_2,\ldots,R_n\}$进行冗余分析并删除其中的冗余规则,若为 permit-overrides 算法,则把所有 $permit$ 类型规则置于 $deny$ 类型规则之前,变为 RS':$\left\{\underbrace{[R^p,...,R^p]}_{permit类型},\underbrace{[R^d,...,R^d]}_{deny类型}\right\}$,且$|RS'|=n' \leqslant n$.同理,deny-overrides 算法下的处理流程类似,first-applicable 算法下,根据算法逻辑不能对规则顺序作出调整.显而易见, $process^{\alpha}_{refine}$ 可以缩减策略规模从而减少匹配运算量,下面着重分析规则顺序调整对策略评估效率的影响.

设定策略 pol 中不含冗余规则且规则数为 n,其中包含 m 个 $permit$ 类型规则$\{R_{p_1},R_{p_2},...,R_{p_m}\}$,$n-m$ 个 $deny$ 类型规则$\{R_{d_1},R_{d_2},...,R_{d_{n-m}}\}$, R_{p_i} 在 pol 中位置表示为$seq(R_{p_i})$.构造 n 个请求,分别匹配且仅匹配 pol 中的一个规则,则易推得完成这 n 次请求所需的 $permit$ 类型规则匹配运算次数为 $\sum_{i=1}^{m} seq(R_{p_i})$, $deny$ 类型规则匹配运算次数为 $(n-m)n$,匹配运算总数表示为 $M=\sum_{i=1}^{m} seq(R_{p_i})+(n-m)n.seq$ 在极端情况下(即 seq 的最后 m 个规则都是 $permit$ 类型)的 M 值取最大,此时,$M_{\max}=(n-m)n+(2n-m+1)m/2$.规则精化后,$seq'$中 $deny$ 类型规则匹配运算次数仍为 $(n-m)n$,但前 m 个规则都是 $permit$ 类型,正好使 M 值最小,此时,$M_{\min}=(m+1)m/2+(n-m)n$,$M_{\max}$ 和 $M_{\min}$ 之间相差 $(n-m)m.seq'$针对结果为 $permit$ 的访问请求,可有效降低规则匹配运算量;针对结果为 $deny$ 的访问请求,任意调整

deny 类型规则的位置都不会增加其规则匹配运算量.

规则精化作为一种判定前的策略预处理技术,与策略匹配评估的具体流程相对独立,因此可作为共性优化技术在多种评估引擎中使用,第 4 节将通过仿真实验分析其优化效率.

3.3 多级缓存机制

策略评估引擎完成一次请求判定需要多种信息的集成处理,因此,处理复杂流程是造成系统延迟的一个重要原因.有效的缓存机制可以降低系统部件之间的交互次数,减少集成处理的代价,避免相关信息的重复检索.本节介绍 MLOBEE 系统采用的多级缓存机制,分别是:判定结果缓存、属性缓存和策略缓存;策略缓存中采取的两阶段策略索引技术将进一步提升策略匹配效率.

3.3.1 判定结果缓存

判定结果缓存(result cache)是加速评估过程最直接的缓存优化机制,即把用户之前的访问结果进行保存,当再次访问时,不必触发属性检索和策略匹配等造成系统响应延迟的复杂流程.用户在访问期内可以激活多个访问会话,在同一会话内可以访问多条具体资源并有不同的判定结果.因此,判定结果缓存应按照主体标识↔会话标识、会话标识↔资源访问结果二层映射模式构建.图 3 是判定结果缓存结构示意图.用户标识 *Principal* 由用户 ID 和其所在域 *Domain* 组成,可区分域内/域外用户;每个用户标识对应一个 *Session ID* 列表,保存该用户激活的所有访问会话;每个 *Session ID* 对应一个访问列表,保存用户在该会话内访问的具体资源 *ResID* 和相应的判定结果 *Result*.Result cache 的设计目标是尽可能地减少相同访问路径引起评估引擎重复运算,作为 MLOBEE 系统的一级缓存,其前端直接面对访问量频繁的用户并发操作请求,后端要考虑授权策略和属性信息的变更问题.因此,应将缓存空间控制在一定规模并制定均衡的缓存有效期.缓存中具体的匹配条目是 *ResID*|*Result* 对,*Result* 的结构形如“*read*:*permit*”.*Result* 还可以通过“*read*:*exception*”和“*read*:*error*”等结构记录判定异常、判定无效等信息,使评估引擎预先识别垃圾请求和无效请求,从而提高抵御恶意 DoS 攻击的能力,减轻评估引擎的处理负担.

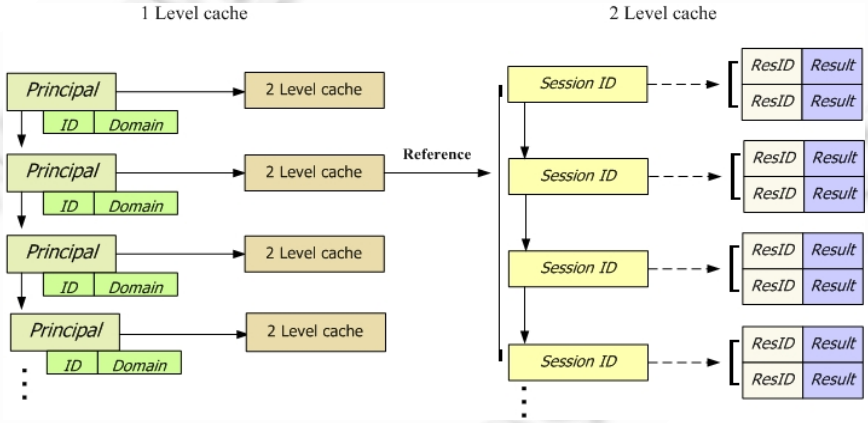


Fig.3 Structure diagram of result cache

图 3 结果缓存结构示意图

3.3.2 属性缓存

属性检索是评估引擎完成策略匹配的必要环节,用户属性的完整程度直接影响策略匹配的成功率.一般的评估引擎可以支持两种最通用的属性传递技术:push 模式和 pull 模式.若采用 push 模式,用户将自身相关属性和请求资源一起发送给评估引擎进行判定.这种方式的优点是简化了策略判定实施过程,缺点是增加了用户的访问负担,属性可信性较低.若采用 pull 模式,评估引擎根据策略匹配具体需要对属性库进行实时检索,这种方式的优点是可信第三方管理的属性安全级别高,缺点是增加了判定引擎通信交互负担.MLOBEE 的属性缓存(attribute cache)机制采取折中方案,用户访问时无须提供自身属性,系统在实施策略匹配前首先检索 Attribute cache,若没有检索结果,则表示该用户是首次发出访问请求,然后通过属性断言服务检索用户属性信息并存入

Attribute cache,后继的访问判定过程直接在属性缓存中对该用户属性进行快速检索.如图 4 所示,属性缓存同样采用二层映射模式,主体标识 *Principal* 和属性标识 *AttrName* 列表组成第 1 层映射,每个 *AttrName* 对应一个属性值列表 *ValueList* 组成第 2 层映射.某些主体属性在系统运行期间变更相对频繁,属性缓存为每个属性标识 *AttrName* 附加一个时间戳 *Time stamp*,以记录其最新加载时间,通过比较时间戳和缓存有效期,定期动态更新过期属性值.Attribute cache 既免除了用户携带大数据量属性信息访问的负担,又规避了频繁远程属性检索的延迟,为策略判定提供基于本地的高效属性查询代理.

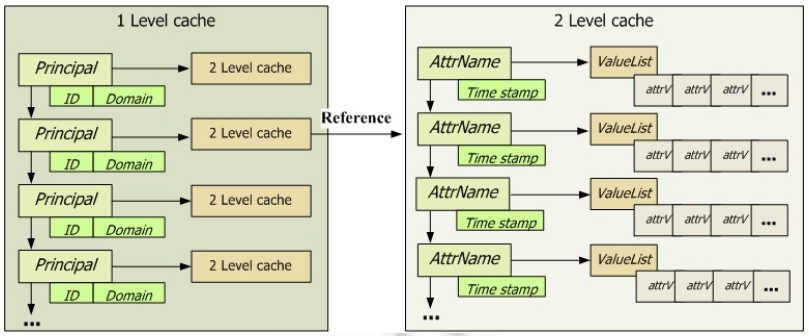


Fig.4 Structure diagram of attribute cache

图 4 属性缓存结构示意图

3.3.3 策略缓存

XACML 策略结构的复杂性导致评估引擎必须对从策略树根节点到叶子节点路径上的所有 Target 进行遍历式匹配运算,缺乏高效策略索引和匹配逻辑,导致每条访问请求都要触发检索大量策略条目和规则条目.文献[22]介绍了 XACML 策略应用的两种主要模式:(1) 多策略组合:多管理方对共同拥有的资源制定个性化的安全策略,策略组合体现为多条 *Policy* 组合为一个 *PolicySet*;(2) 多规则组合:组织内制定针对多个用户和多种资源的安全策略,策略组合体现为多条 *Rule* 组合为一个 *Policy*.MLOBEE 系统针对上述应用特点,通过两阶段索引技术实现策略缓存机制(policy cache).

第 1 阶段索引针对多策略组合,考虑到安全策略的实施最终要落实到资源信息的保护,首先从策略目标中提取资源属性条目 *res* 作为索引主键,每个主键指向一个策略列表 *policyList*.策略加载时,将每条策略添加到其包含资源属性所指向的策略列表中,形成哈希表 *HashTable*(*res*,*policyList*),这称为 MLOBEE 策略缓存的第 1 阶段索引.第 1 阶段索引借鉴了策略目标的交集计算方法,策略列表内的策略都包含针对相同 *res* 索引的权限定义;另外,对于存在层次关系的资源属性,所有上层资源的策略都自动适用其蕴含的下层资源.通过这种资源定位方式,评估引擎可以在不进行完全匹配的情况下,迅速缩减实际评估的策略规模.

第 2 阶段索引针对策略内多个规则的目标元素,其目标是通过访问请求中提取的部分信息快速定位一个较为精确的检索子集,且该检索子集为扁平式结构,其内部信息描述简单,检索代价的上限即对子集中所有元素进行匹配所需的耗时,该过程完成后即给出明确的评估结果.首先按规则精化后的规则顺序,依次从每条规则的目标元素中提取资源属性子集与动作属性子集的笛卡尔积,其逻辑表达式为 $PerUnit=(Target.res)\otimes(Target.ac)$,称 $perAtom(res,ac)\in PerUnit$ 为权限原语;然后依次将 $perAtom(res,ac)\in PerUnit$ 所在目标元素内的主体属性分别组成主体属性表 *subList*,*subList* 中的每个主体属性都附带所在规则的 *effect*,形如 $subAttr^{effect}$;以权限原语为索引建立 *perAtom* 到主体属性表的映射结构,形如哈希表 *HashTable*(*perAtom*,*subList*),这称为策略缓存的第 2 阶段索引.该索引改变了策略内以规则为单位的嵌套式组织结构,形成以权限原语为索引的扁平式描述结构,将策略内的规则匹配逻辑统一为首次适用匹配逻辑.图 5 是针对图 1 实现的策略缓存示意图.为了避免策略实体在缓存中重复加载,策略列表中只存放策略标识,其指向内存中唯一的策略实体,权限单元中的属性值用其在策略定义中的数字下标表示.下面详细说明第 2 阶段索引建立过程.

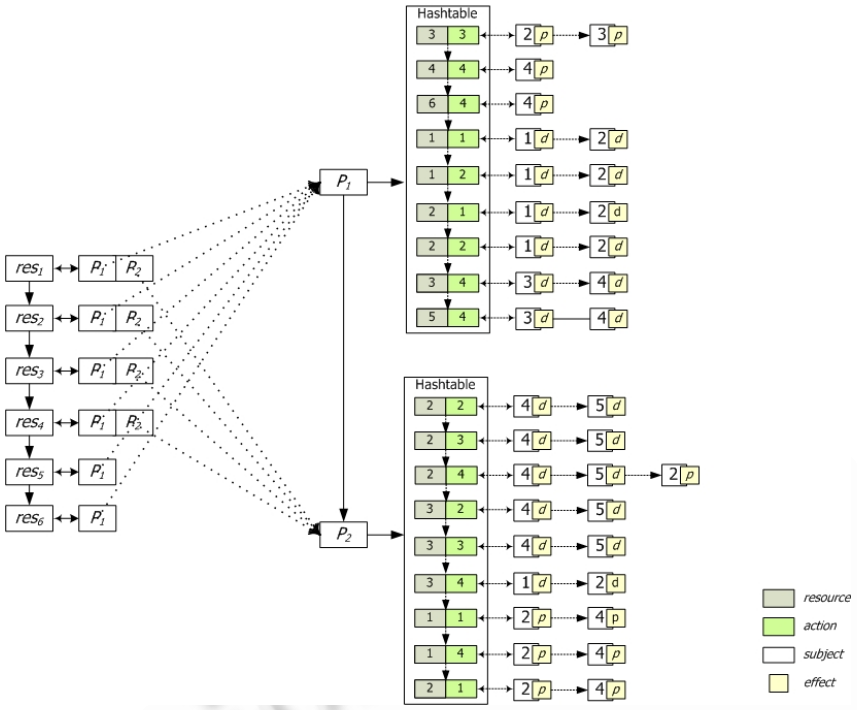


Fig.5 Structure diagram of policy cache

图 5 策略缓存结构示意图

策略 P_1 内的 4 条规则目标简化表示为 T_1={[sub_1,sub_2],[res_1,res_2],[ac_1,ac_2]}^{d},T_2={[sub_2,sub_3],[res_3],[ac_3]}^{p},T_3={[sub_3,sub_4],[res_3,res_5],[ac_4]}^{d},T_4={[sub_4],[res_4,res_6],[ac_4]}^{p};策略 P_2 内的 4 条规则目标简化表示为 T_5={[sub_2,sub_4],[res_1,res_2],[ac_1,ac_4]}^{p},T_6={[sub_4,sub_5],[res_2,res_3],[ac_2,ac_3,ac_4]}^{d},T_7={[sub_4],[res_2],[ac_1,ac_3]}^{p},T_8={[sub_2],[res_1],[ac_1]}^{p}.P_1 的规则合并算法是 Permit-Overrides.因此,规则精化流程后的规则排列顺序将变为 R_2→R_4→R_1→R_3,即 $seq(T_2)$<$seq(T_4)$<$seq(T_1)$<$seq(T_3)$.首先从 T_2 中提取 $PerUnit$=(res_3)⊗(ac_3),其中仅包含 1 条权限原语 $perAtom$=(res_3,ac_3),将 $perAtom$(res_3,ac_3)↔$subList$(sub_2^p,sub_3^p)添加到第 2 阶段索引中.然后,依次对 T_4,T_1,T_3 建立索引项.策略 P_2 的规则精化后的顺序为 R_6→R_5→R_7→R_8,其缓存建立过程需要注意几个问题:首先,经过规则冗余分析,规则 R_8 针对 R_5 是冗余规则,因此规则精化过程将其删除,策略缓存初始化不再导入该规则;其次,T_5∩T_6={[sub_4],[res_2],[ac_4]},且考虑到规则评估合并算法是 Deny-Overrides,可推得 R_5 中 sub_4:{res_2,ac_4}^{p} 相对于 R_6 中 sub_4:{res_2,ac_4}^{d} 是无效的权限表达,因此在权限原语 $perAtom$(res_2,ac_4)对应的 $subList$ 中只包含 sub_4^p ,不再添加 sub_4^p 条目;最后,虽然 R_7 不是冗余规则,但其拆分出的权限 sub_4:{res_2,ac_1}^{p} 已被 R_5 所涵盖,且权限 sub_4:{res_2,ac_3}^{p} 相对于 R_6 是无效表达,所以 R_7 是无效规则,其导入过程不触发实际的索引添加操作.

将以上过程归纳,得出两阶段索引策略缓存应遵循的完整创建规则如下:

规则 7. 第 1 阶段索引中的属性资源条目 res 全部取自于策略目标 $pol.Target$,若策略目标中省略了对资源属性的约束,则默认该策略适用所有资源条目;下层资源属性指向的策略列表自动添加上层资源属性策略列表中的策略.

规则 8. 第 2 阶段索引从规则目标的资源属性子集与动作属性子集的笛卡尔积中拆分权限原语 $perAtom$(res,ac)为键值,将其所在规则中的主体属性添加到原语指向的主体属性列表 $subList$,建立 $perAtom$(res,ac)和 $subList$ 的映射关系,$subList$ 中主体属性元素都附带其所在规则的 $effect$,形如 sub^{effect}.

规则 9. 对同一策略内的规则 R_i 和 R_j,若 $seq(T_i)$<$seq(T_j)$且 $perAtom$(res,ac)∈$PerUnit_i$∧$perAtom$(res,ac)∈$PerUnit_j$,则对 T_j 中权限原语建立索引时不再重复建立 $perAtom$(res,ac)索引项,而是把 T_j 主体属性子集 $subSet_i$

中的元素直接添加到该原语先前已建立的 *subList* 中.

规则 10. 若{*sub*,*res*,*ac*}$\in T_i \cap T_j$,则:当 $effect_i=effect_j$ 时,在 *perAtom*(*res*,*ac*)相应 *subList* 中添加 *sub* 的操作只进行 1 次;当 $effect_i \neq effect_j$ 时,根据规则评估合并算法的优先匹配类型 *prefer-effect*(*permit* 或 *deny*),只把 $sub^{prefer\text{-}effect}$ 添加到 *subList* 中.

规则 11. 对于多规则组合的应用场景,不需要建立第 1 阶段索引,直接进行第 2 阶段索引.

简要分析两阶段索引可显著提高策略匹配效率.设有 m_1 条策略,每条策略包含 m_2 条规则,规则目标内包含的主体属性、资源属性和动作属性数目分别为 n_1,n_2,n_3.因为策略目标取自其包含规则目标的并集,所以匹配一个 $Target_{pol}$ 的时间复杂性大约为 $\Theta(m_2\cdot(n_1+n_2+n_3))$.如果策略目标适合,则还要对内部的规则进行遍历匹配.因此,按照一般的策略评估流程,完成一次访问请求的时间复杂性上限大约是 $\Theta(2m_1\cdot m_2\cdot(n_1+n_2+n_3))$.在两阶段索引方案下,首先从请求 *req*(*subSet*,*res*,*ac*)提取 *res* 并计算 *hashcode*(*res*),通过第 1 阶段索引快速定位 *res* 指向的策略列表 *policyList*,列表长度即为本次请求要匹配的策略总和;然后,从请求中提取 *perAtom*(*res*,*ac*)并计算 *hashcode*(*perAtom*(*res*,*ac*)),通过第 2 阶段索引定位主体属性列表 *subList*,该表长度即为策略内要匹配的属性总和.当一条策略评估完成后,引擎要根据评估结果和评估合并算法决定是否要对 *policyList* 中的后继策略继续匹配,*policyList* 在极限情况下的长度为 m_1.同理,*subList* 在极限情况下的长度为 $m_2\cdot n_1$.设哈希计算的耗时为常量 *T*,则两阶段索引方案时间复杂性的上限大约可表示为 $\Theta(2T+m_1\cdot m_2\cdot n_1)$,其远小于 $\Theta(2m_1\cdot m_2\cdot(n_1+n_2+n_3))$的耗时代价.

3.4 评估判定流程

图 6 给出了 MLOBEE 评估引擎完成一次完整判定的时序流程.

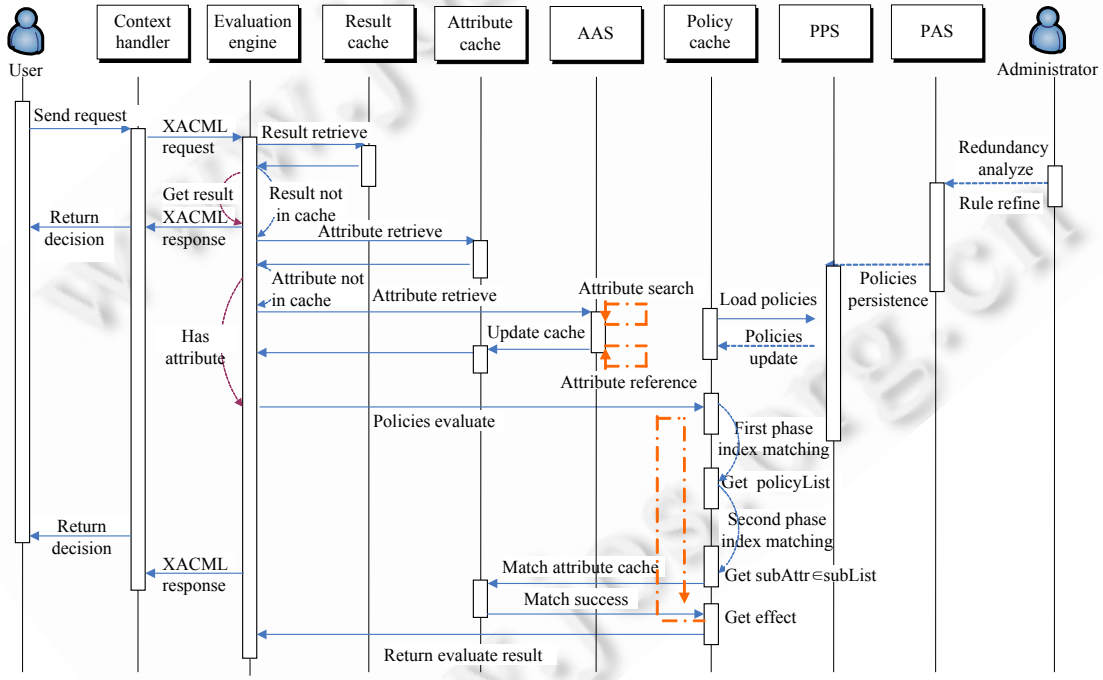


Fig.6 Sequential process diagram of MLOBEE policies evaluating

图 6 MLOBEE 策略评估时序流程图

设定用户发出请求 *req*(*principal*,*res*,*ac*),其中,*principal* 是用户自身标识,*res* 是要访问资源,*ac* 是访问动作.PDS 中的上下文处理器 *ContextHandler* 将请求转换为 *XACMLRequest*(*principal*,*res*,*ac*),评估引擎 *EvaluationEngine* 对 *XACMLRequest* 进行处理,根据用户的 *principal* 和此次会话标识 *Session ID* 在 Result Cache

检索是否有关于(*res*,*ac*)的判定结果.若 Result Cache 中存在判定结果,则引擎直接将其封装为 *XACMLResponse* 返回;若没有检索到判定结果,则引擎首先判断属性缓存中是否有 *principal* 的相关属性,*principal* 首次访问资源时,需要通过属性断言服务 AAS 检索并更新 Attribute Cache.若用户来自外域,还需要推导其在本域的等价属性.若应用场景为多规则组合,则直接进行策略缓存第 2 阶段索引,根据 *perAtom*(*res*,*ac*)获取 *subList*,并将 *subAttr*∈*subList* 按顺序在属性缓存中匹配,首次适用的 *subAttr* 所附带的 *effect* 类型即为该策略的评估结果;若应用场景为多策略组合,则引擎根据 *res* 在策略缓存中进行第 1 阶段索引匹配,获取策略列表 *policyList*,针对其中的每条策略,实施第 2 阶段索引.在多策略组合情况下,将应用策略评估合并算法:若某条策略的评估结果符合算法的优先匹配逻辑,则直接返回该结果;若不符合优先匹配逻辑,则评估引擎继续对 *policyList* 中后继策略进行评估,最终评估结果经过上下文处理器返回用户.

需要说明的是,策略管理服务 PMS 和策略持久化服务 PPS 相对判定评估流程是异步行为模式,主要通过系统管理员触发操作流程.管理员首先通过 PMS 对初始策略进行冗余分析和规则精化,然后将优化后的策略集合通过 PPS 进行持久化存储,Policy Cache 则通过 PPS 提供的服务接口将策略加载到内存中,当修改、创建、删除等操作触发策略状态改变时,PPS 将及时通知 Policy Cache 更新.

4 技术比较及性能分析

本节从策略加载方式和策略匹配模式的角度分析各种评估引擎的技术原理差异,在此基础上,通过多种类型的仿真测试,验证 MLOBEE 多层次优化技术的有效性和其整体评估性能优势.

4.1 策略加载方式及策略匹配模式比较

通过调研比较多种判定引擎系统的具体实现可知,评估优化技术的差异主要集中在策略加载模式和策略匹配模式方面.

根据策略加载内存的时机,可分为动态加载和静态加载.采用动态加载的评估引擎在接收到访问请求后,实时地从策略库中检索并取回适用策略;静态加载方式则发生在评估引擎初始化阶段,系统一次性地将可用策略全部加载到内存中,此后的策略评估都在内存中进行.动态加载的优点是节省内存空间,引擎直接匹配的策略规模有限,策略提供方式灵活;缺点是增加了系统通信开销,策略匹配性能取决于策略库的检索方式和效率.静态加载方式无疑会增加系统的内存开销,但如果能在内存中建立高效的策略索引结构并辅以相应的评估优化算法,则内存运算的速度优势会大幅度提升评估效率.

策略匹配模式方面,大多数以 Sun XACML 作为引擎核心模块的系统(例如 JBoss XACML 和 Melcoe PDP)基本上采用二次匹配的方式对访问请求进行评估:首先在内存的策略加载区通过对策略目标的初次匹配获取适用策略,将多条适用策略组成策略集返回给引擎,若没有适用策略则返回异常信息,引擎不再进一步匹配;引擎针对返回的策略集进行二次详细匹配,包括策略目标匹配和策略内的规则目标匹配,结合具体合并算法深度优先遍历策略对象.JBoss XACML 策略处理模块比较简单,通过配置文件指定策略存储位置并采用静态加载策略方式;Melcoe PDP 则将二次匹配扩展为三次匹配模式:先将策略文件存储在 XML Native 数据库中,通过属性约束查询完成一次匹配,将获取的策略子集动态加载内存,然后针对策略目标二次匹配,进一步缩减策略规模,第 3 次匹配完成对访问请求的完整评估并返回最终结果.Melcoe 系统的特色在于,借助 XML 类型数据库专有的查询优化和索引优化来提高策略匹配效率,但其在缩小策略空间的同时增加了匹配层次,必须考虑由此带来的时间延迟.

另外,策略在内存中的存储结构对评估引擎的策略匹配模式也有很大的影响.JBoss 引擎和 Melcoe 引擎在策略加载内存后都采用列表嵌套或集合嵌套作为存储结构,XACMLight 和 AXESCON XACML 没有采用 Sun 的开发包,都是独立实现评估引擎,但其策略存储方式与前两者类似,匹配逻辑大体相同.AXESCON XACML 在首次匹配时对策略目标进行了完整的评估,二次匹配时直接对规则目标进行评估.Enterprise XACML 策略存储方式与上述系统有明显的差别,这也是其声称评估效率更佳的主要原因.该引擎将策略目标分解为若干属性标识,作为索引项,每个策略都与自身包含的标识建立索引,系统在将策略载入内存时,实际建立了一个属性实体

标识和策略间的多对多映射.虽然该系统仅对策略目标建立了索引结构,但与其他采用二次匹配的系统相比,其基于 HashMap 的索引结构使得策略匹配的速度显著提高.XEngine 系统在策略评估前将策略规则转化为基于整数范围的规则,并将 XACML 策略分层嵌套结构转换为单层结构,相应的匹配逻辑也随之简化.按照其提供的理论分析,XEngine 的评估效率应该有显著的提升,但在本文完成之前,该系统还没有开源代码和详细技术资料,无法分析引擎代码实现流程并搭建仿真测试环境,因此,后面的分析比较和仿真实验未予考虑**.

与以上引擎相比,MLOBEE 系统最大的特点是多层次优化技术:功能部件方面,MLOBEE 在实施访问判定前引入了规则精化技术,在不影响引擎判定结果的情况下删除冗余规则并调整规则序列;评估引擎借助多级缓存机制,节省了大量的系统交互开销;策略存储方面,MLOBEE 采用两阶段索引技术实现策略缓存,将嵌套式匹配逻辑转化为偏平式匹配逻辑,缩减实际匹配的策略空间并简化匹配过程.另外,MLOBEE 针对外域用户访问引入属性推理引擎,根据用户外域属性和推理规则导出等价的域内属性,无须额外制定针对域外陌生属性的策略,降低了授权管理的风险和规模.表 1 给出了各种评估引擎的技术细节比较.

Table 1 Comparison of evaluation engine technologies

表 1 评估引擎技术比较

	Policy loading mode	Policy storing mode	Policy matching pattern	Special optimization techniques
Sun XACML	Static loading or dynamic loading	Policy list	Twice matching	None
Jboss XACML	Static loading	Policy list	Twice matching	Policy configuration file
Melcoe PDP	Dynamic loading	Policy list	Thrice matching	Attribute constraint, native XML database indexing
Xacmlight	Static loading	Policy list	Target separation matching	None
AXESCON XACML	Dynamic loading	Policy list	Twice matching	Second matching omitting policy target
Enterprise XACML	Static loading	One-Level hashtable	Once searching+ once matching	Policy indexing based on attribute entry
XEngine	Unknown	Unknown	Unknown	Policy numericalization, policy normalization
MLOBEE	Static loading	Two-Level hashtable	Once matching in policy	Rule refining, multi-level cache, two-stage policy indexing

4.2 仿真实验分析

仿真测试的目的是比较目前各类引擎的评估性能,给出量化实验结果,尤其是针对 MLOBEE 中采用的多层次优化技术逐一构造测试用例,通过量化分析进行技术验证.仿真实验涉及的评估引擎包括 JBoss XACML, Melcoe PDP,AXESCON XACML,Enterprise XACML 和 MLOBEE.我们主要从以下几方面进行测试分析:(1) 规则精化作为共性优化技术对多种引擎的性能提升;(2) 判定结果缓存优化性能测试;(3) 属性缓存减少系统交互开销的有效性;(4) 各种策略存储技术的性能差异;(5) 各系统的综合评估性能.仿真测试需要的策略条目都是在 XACML 策略一致性测试包[26]基础上,根据具体实验场景进行修改和扩充.实验环境为:Intel Pentium4 3.0GHz 双核 CPU,1.5G 内存,Windows XP SP2 操作系统平台,Java Runtime Environment 1.6.10.

4.2.1 规则精化性能分析

该实验将规则精化后的策略集合提供给 5 种引擎,比较规则精化前后的判定性能差异,目的是体现该优化技术对多种引擎的通用性.由于各评估引擎的策略加载方式和策略匹配模式不尽相同,因此对每种引擎进行纵向比较,计算规则精化前后平均处理一次请求的耗时差别.实验提供 3 组用例样本:第 1 组由 500 条原始策略组成,其中包含 100 个冗余规则,规则精化后需要对内部规则重新排序的策略有 200 条;第 2 组由 1 000 条原始策略组成,其中包含 300 个冗余规则,规则精化后需要对内部规则重新排序的策略有 400 条;第 3 组由 2 000 条原始策略组成,其中包含 500 个冗余规则,规则精化后需要对内部规则重新排序的策略有 800 条.以上 3 组样本中

** XEngine 的源码和技术资料现已公开(发布时间为 2010 年 8 月 5 日,项目网址为 http://xacmlpdp.sourceforge.net/).

的每条策略平均包含 3 条规则,根据策略中包含的各种类型属性值,分别随机生成 1 000 次不同的访问请求(请求中包含完整属性信息,不再需要属性检索操作),各评估引擎首先根据原始策略用例对访问请求进行评估,然后根据规则精化处理后的策略集合评估访问请求.

如图 7 所示,在 3 组策略样本测试下,JBoss XACML 判定速度分别提升了 30%,30%,24%;Melcoe PDP 分别提升了 17%,16%,15%;AXESCON XACML 分别提升了 15%,14%,12%;Enterprise XACML 分别提升了 12%,9%,8%;MLOBEE 分别提升了 11%,8%,7%.前 3 种引擎通过规则精化带来的效率提升是比较明显的;Enterprise XACML 和 MLOBEE 由于都采用了策略索引优化,在一定程度上缓解了策略检索效率对策略规模的正相关依赖,因此效率提升指标稍低.

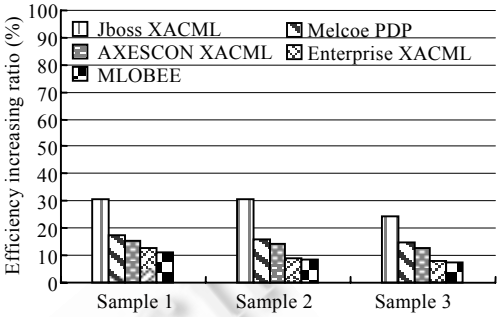


Fig.7 Performance comparison of rule refining

图 7 规则精化性能分析

4.2.2 判定结果缓存性能分析

各引擎对相同请求的首次评估都需要进行实质的策略匹配和属性查询等操作,但对于其后的重复请求,应用了 Result cache 相关技术的引擎则会显示其性能优势.为了准确地分析判定结果缓存对整个系统的影响,实验尽可能地屏蔽其他技术细节(例如,策略存储结构、匹配模式以及属性检索)差异对评估速度的影响,提供的 3 组策略样本都是多规则组合的二层描述结构,即多条规则组成唯一策略,这种简化的策略结构最大程度上限制了不同策略存储优化技术和策略匹配优化技术造成的性能分析误差.3 条经过规则精化处理后的策略样本分别由 1 000,2 000,4 000 条规则组成,分别构造 100,200,400 个不同请求,每个请求连续发送 100 次,除去每种请求的首次响应时间,计算各评估引擎完成一次响应的平均耗时.

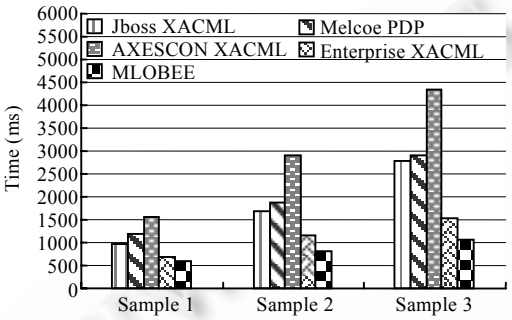


Fig.8 Performance comparison of evaluation result cache

图 8 评估结果缓存性能比较

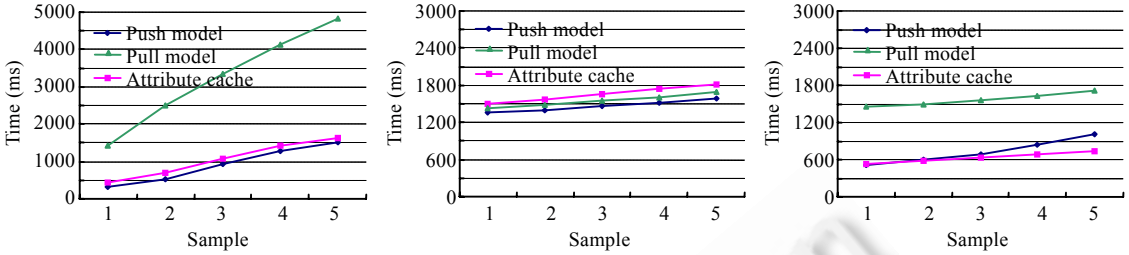
图 8 中的测试结果显示,在采用 Result Cache 技术的 Enterprise XACML 和 MLOBEE 在处理短时间内重复请求的场景时,评估效率优势明显.针对策略样本 3,Enterprise XACML 分别比 JBoss XACML,Melcoe PDP 和 AXESCON XACML 提高了 81%,90%,183%;MLOBEE 则分别比后三者提高了 162%,174%,308%.

4.2.3 属性缓存相关测试分析

本实验针对 MLOBEE 系统,分别测试 pull 模式、push 模式、Attribute Cache 模式下属性检索的响应时间.为 3 种应用场景各设计 5 组测试样本:(1) 单用户多次请求:相同用户分别请求 20,40,60,80,100 次,每次请求属性个数为 5,10,15,20,25;(2) 多用户单次请求:10,20,30,40,50 个 *principal* 发出单次属性请求,每次请求的属性个数为 5;(3) 多用户多次请求:10,20,30,40,50 个 *principal* 分别请求次数为 20,40,60,80,100,每次请求的属性个数为 5.引擎调用 AAS 的属性查询服务接口,通过 SAML 协议传递安全属性断言,计算评估引擎从发出属性请求到获取属性值的平均耗时.

图 9(a)显示,在单用户多次请求场景下,push 模式属性检索性能最好,*principal* 只在首次访问时进行一次属性查询并将属性信息封装在访问请求中,Attribute cache 模式下每次属性查询耗时和 push 模式的差别并不明显,原因在于其高效的检索结构,pull 模式由于每次请求都须调用远程服务,与另外两者的延迟差距显著.图 9(b)显示,在多用户单次请求场景下,各种模式的属性查询性能近似,其原因在于每次请求的 *principal* 都是首次访问,Attribute cache 模式由于要进行缓存更新,时间延迟反而稍长.图 9(c)显示,在多用户多次请求场景下,随着

principal 规模的增长和请求次数的增加,Attribute cache 模式的查询性能逐渐稍优于 pull 模式,属性缓存定位所使用的处理器时钟数小于 XML 解析所用时钟数,其存储结构更有利于处理并发请求.



(a) One user, several requests (a) 单用户多次请求 (b) Many users, single request (b) 多用户单次请求 (c) Many users, several requests (c) 多用户多次请求

Fig.9 Analysis of attribute cache related tests

图 9 属性缓存相关测试分析

可以看出,Attribute cache 模式的检索性能最稳定,适合处理大规模资源访问的应用场景;另外,属性查询平均耗时随属性查询个数有所增加.

4.2.4 策略优化相关测试分析

采用第 4.2.1 节中规则精化后的策略样本和请求用例,屏蔽判定结果缓存和属性缓存对评估过程的影响.从图 10 中的数据可以看出 MLOBEE 引擎的明显优势:在 3 组样本下,判定速度分别比 JBoss XACML,Melcoe PDP, AXESCON XACML 和 Enterprise XACML 提高了(22%,49%,104%,19%),(84%,96%,220%,29%)和(122%,137%, 261%,33%).在大规模策略匹配方面,JBoss XACML 和 Melcoe PDP 表现居中,评估性能比较稳定;AXESCON XACML 与其他几种引擎的性能差距比较明显,其原因在于,动态加载策略方式针对大规模策略场景的策略检索耗时较大;Enterprise XACML 和 MLOBEE 由于加强了策略索引优化,在性能测试中表现最好.

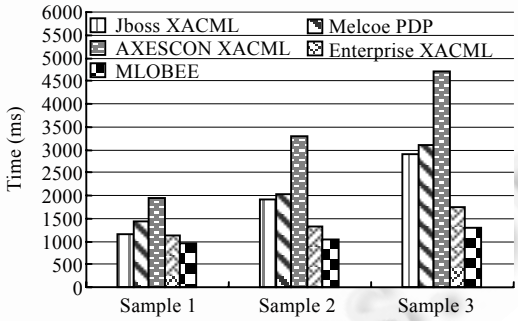


Fig.10 Performance comparison of policy optimization

图 10 策略优化性能比较

4.2.5 引擎综合评估性能分析

测试各引擎整体评估性能的差异,需要完整地实现每种引擎的自身技术细节和特有的优化方案,对表 1 中列出的评估引擎技术进行综合衡量.为了检验 MLOBEE 在多规则组合、多策略组合不同场景下的实际性能,需要准备两组策略样本:第 1 组由 8 000 条规则组成单一策略,并且包含 1 000 条冗余规则,需要前提位置的规则有 2 000 条;第 2 组由 3 000 条策略组成,平均每条策略包含 3 条规则,整个样本存在 2 000 条冗余规则,需要前提位置的规则有 4 000 条.为 50 个用户标识分别构造 500 个不同的访问请求,每个请求共随机发送 5 次,每个标识平均拥有 5 个属性值.需要注意的是,除了 MLOBEE 系统以外,其他引擎使用的策略库都没有经过规则精化处理,只有 Enterprise XACML 和 MLOBEE 使用了 Result Cache 技术,只有 MLOBEE 使用了 Attribute cache 技术.

由图 11 可以看出,虽然两组样本的规则总体规模近似,但各种引擎对多规则组合样本的评估速度普遍快于多策略组合样本,这主要是由后者的策略结构比较复杂所导致,引擎在策略解析匹配方面占用更多的处理时间.在样本 1 情况下,MLOBEE 处理一次请求的平均速度分别比 JBoss XACML,Melcoe PDP,AXESCON XACML 和 Enterprise XACML 提高 345%,434%,502%,104%;在样本 2 情况下,平均速度分别提高 192%,255%,311%,51%.

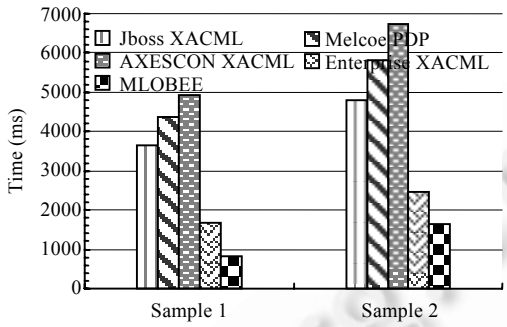


Fig.11 Evaluation engine's overall performance comparison

图 11 评估引擎整体性能比较

5 结 论

本文针对 XACML 判定评估问题分析了现有系统的策略匹配技术和评估原理,提出一种实现多层次优化的策略判定评估引擎 MLOBEE:在判定评估前对策略进行预处理,通过规则精化缩减策略规模并调整规则顺序,实现前期优化;判定过程中实施包括判定结果缓存、属性缓和策略缓存的多级缓存机制,节省了系统交互开销;策略缓存内部通过两阶段索引技术实现策略存储,从而将嵌套式匹配逻辑优化为扁平式匹配逻辑,提高了策略匹配效率.通过详细的仿真测试分析,验证了 MLOBEE 多层次优化技术的有效性,其整体评估性能明显优于大多数现有评估引擎.

References:

[1] XACML reference. 2007. http://docs.oasis-open.org/xacml/references/xacmlRefsV1.83.html

[2] Moses T. Extensible access control markup language (XACML) version 2.0. Technical Report, OASIS Standard, 2005.

[3] Fisler K, Krishnamurthi S, Meyerovich LA, Tschantz MC. Verification and change-impact analysis of access-control policies. In: Proc. of the 27th Int'l Conf. on Software Engineering. New York: ACM Press, 2005. 196−205. [doi: 10.1145/1062455.1062502]

[4] Kolovski V, Hendler J, Parsia B. Analyzing web access control policies. In: Proc. of the 16th Int'l Conf. on World Wide Web. New York: ACM Press, 2007. 677−686. [doi: 10.1145/1242572.1242664]

[5] Martin E, Xie T, Yu T. Defining and measuring policy coverage in testing access control policies. In: Proc. of the 8th Int'l Conf. on Information and Communications Security. Berlin: Springer-Verlag, 2006. 139−158.

[6] Mazzoleni P, Bertino E, Crispo B, Sivasubramanian S. XACML policy integration algorithms: not to be confused with XACML policy combination algorithms! In: Proc. of the 11th ACM Symp. on Access Control Models and Technologies. New York: ACM Press, 2006. 219−227. [doi: 10.1145/1133058.1133089]

[7] Tschantz MC, Krishnamurthi S. Towards reasonability properties for access-control policy languages. In: Proc. of the 11th ACM Symp. on Access Control Models and Technologies. New York: ACM Press, 2006. 160−169. [doi: 10.1145/1133058.1133081]

[8] Guelev DP, Ryan M, Schobbens PY. Model-Checking access control policies. In: Zhang K, Zheng Y, eds. Proc. of the ISC 2004. LNCS 3225, Berlin: Springer-Verlag, 2004. 219−230. [doi: 10.1007/978-3-540-30144-8_19]

[9] Bryans J. Reasoning about XACML policies using CSP. In: Proc. of the 2005 Workshop on Secure Web Services. New York: ACM Press, 2005. 28−35. [doi: 10.1145/1103022.1103028]

[10] Lin D, Rao P, Bertino E, Lobo J. An approach to evaluate policy similarity. In: Proc. of the 12th ACM Symp. on Access Control Models and Technologies. New York: ACM Press, 2007. 1−10. [doi: 10.1145/1266840.1266842]

[11] Sun XACML. 2006. http://sunxacml.sourceforge.net/

[12] XACML.NET. 2005. http://mvpos.sourceforge.net/index.html

[13] Parthenon XACML. 2005. http://www.parthcomp.com/xacml_toolkit.html

[14] JBoss XACML. 2008. http://www.jboss.org/jbosssecurity/download/index.html

[15] Melcoe PDP. 2008. http://www.muradora.org/muradora/wiki/MelcoePDPDoc

[16] XACMLight. 2008. http://sourceforge.net/projects/xacmllight/

[17] AXESCON XACML. 2006. http://axescon.com/ax2e/

[18] Enterprise XACML. 2008. http://code.google.com/p/enterprise-java-xacml/

[19] Liu AX, Chen F, Hwang JH, Xie T. XEngine: A fast and scalable XACML policy evaluation engine. In: Proc. of the 2008 ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems. New York: ACM Press, 2008. 265−276. [doi: 10.1145/1375457.1375488]

[20] Li N, Hwang JH, Xie T. Multiple-Implementation testing for XACML implementations. In: Proc. of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications. New York: ACM Press, 2008. 27−33. [doi: 10.1145/1390832.1390837]

[21] Turkmen F, Crispo B. Performance evaluation of XACML PDP implementations. In: Proc. of the 2008 ACM Workshop on Secure Web Services. New York: ACM Press, 2008. 37−44. [doi: 10.1145/1456492.1456499]

[22] Li XF, Feng DG, Xu Z. Access control policy management based on extended-XACML. Journal on Communications, 2007,28(1): 103−110 (in Chinese with English abstract).

[23] Li XF, Feng DG, He YZ. Research on preprocessing policies in XACML admin. Journal of Computer Research and Development, 2007,44(5):729−736 (in Chinese with English abstract). [doi: 10.1360/crad20070501]

[24] Nie XW, Feng DG. TXACML—An access control policy framework based on trusted platform. Journal of Computer Research and Development, 2008,45(10):1676−1686 (in Chinese with English abstract).

[25] Wang YZ, Feng DG. A conflict and redundancy analysis method for XACML rules. Chinese Journal of Computers, 2009,32(3): 516−530 (in Chinese with English abstract)

[26] XACML 2.0 conformance tests. 2005. http://www.oasis-open.org/committees/download.php/14846/xacml2.0-ct-v.0.4.zip

附中文参考文献:

[22] 李晓峰,冯登国,徐震.基于扩展 XACML 的策略管理.通信学报,2007,28(1):103−110.

[23] 李晓峰,冯登国,何永忠.XACML Admin 中的策略预处理研究.计算机研究与发展,2007,44(5):729−736. [doi: 10.1360/crad20070501]

[24] 聂晓伟,冯登国.基于可信平台的一种访问控制策略框架——TXACML.计算机研究与发展,2008,45(10):1676−1686.

[25] 王雅哲,冯登国.一种 XACML 规则冲突及冗余分析方法.计算机学报,2009,32(3):516−530.

王雅哲(1979－),男,山东济南人,博士,助理研究员,主要研究领域为信息系统安全,分布式计算.

张立武(1976－),男,博士,高级工程师,主要研究领域为安全体系架构,认证协议,授权访问控制.

冯登国(1965－),男,博士,研究员,博士生导师,主要研究领域为密码学,信息安全.

张敏(1975－),女,博士,高级工程师,CCF 会员,主要研究领域为数据库安全技术,系统安全技术.