一个灵活的操作系统安全框架 FMAC*)

易晓东　杨学军
(国防科大计算机学院　长沙 410073)

摘　要　当前,操作系统的安全需求越来越多样、灵活和具体,它们往往只描述系统中一小部分实体之间的约束,但对安全策略的灵活性及定制的简洁性有较高的要求。传统的安全框架,如 FLASK 等,难于满足此类"轻量级"的安全需求。本文提出的 FMAC 框架专门面向此类安全需求,它定义了基于标记迁移系统 LTS 的安全策略模型,以规范和简化安全策略的定制。FMAC 框架由通用的对象管理器模型和安全策略管理器模型组成。讨论了 FMAC 在 Unix 类操作系统中的实现,通过层次式的客体组织与基于角色的主体组织,普通用户可以方便快捷地定制出满足要求的轻量级安全策略。

关键词　FMAC,操作系统安全,安全策略模型,强制访问控制框架

FMAC: A Flexible Security Framework for Secure OS

YI Xiao-Dong　YANG Xue-Jun
(College of Computer, National Univ. of Defense Technology, Changsha 410073)

Abstract　The security requirements of current operating system become more and more diverse, flexible and concrete. Usually, they only specify the constraints among a little set of entities of the system, but expect to be implemented quickly and expediently. These "lightweight" security requirements can't be satisfied well in traditional security framework like FLASK and this is why FMAC is presented. The FMAC model is made up of two sub-models named object manager and security policy manager. FMAC also defines a security policy model based on labeled transition system (LTS) in order to standardize and facilitate the policy customization. How to implement FMAC in Unix-class operating systems is also discussed. It's simple to customize the lightweight security policies with the hierarchical structure of objects and role based structure of subjects which are two key elements of the implementation.

Keywords　FMAC, Operating system security, Security policy model, Mandatory access control framework

1　引言

不同的计算环境和不同的应用对操作系统有不同的安全需求,因此操作系统必须支持多安全策略以及灵活可变的安全策略来满足不同的安全需求,FLASK 就是这样一个操作系统安全框架。FLASK[2]基于强制访问控制 MAC(Mandatory Access Control)实现,支持多安全策略和灵活可变的安全策略,是操作系统安全研究的最著名成果之一。

现在的操作系统安全需求越来越多样化,也越来越灵活、具体和细腻。例如,一个普通用户从网上下载了一个不可信的程序,为了安全,他可能需要对这个程序的功能有所约束,比如不允许该程序修改该用户的程序和数据等。操作系统除了支持施加于整个系统范围以约束整个系统行为的"重量级"安全策略外,还要提供机制,供普通用户灵活、方便地定制安全策略以满足这些"轻量级"的安全需求。此类的安全需求有三个特点:一是该安全需求所涉及的主体与客体往往只是系统所有主客体集合的一小部分;二是此类安全策略数量多,灵活度高,经常被加载和卸载;三是此类安全策略往往比较简单,但要求可以简洁方便地建立和加载与卸载,另外还要求普通的用户可以建立安全策略,而不需要专门的知识,也不需要编写程序或编写复杂的安全配置文件。

当前的操作系统安全框架,包括 FLASK,只支持施加于整个操作系统所有主客体的安全策略,而且其安全策略通过程序代码或复杂的配置文件实现,不利于方便快速地建立新的安全策略,同时限制了操作系统可以施加的安全策略数量,因而不能满足当前操作系统安全策略的要求。针对这些问题和当前的安全需求,我们借鉴了 FLASK 的成功思想,设计了一种适合于当前安全需求的灵活强制访问控制框架 FMAC (Flexible MAC)。FMAC 支持只施加于系统部分主客体的安全策略,并提供可视化支持,使得用户可以不用书写任何代码和配置文件,就可以方便简洁地定制新的安全策略。

为了规范和简化安全策略的定制,FMAC 首先定义了通用的安全策略模型。该安全策略模型基于标记迁移系统 LTS(Labeled Transition System)定义,描述为状态机形式,可以基于该模型以可视化的形式方便快捷地定制出当前操作系统使用的所有访问控制和信息流类的安全策略。FMAC 框架自身由对象管理器和安全策略管理器两部分组成,对象管理器负责管理操作系统中动态生成的所有主客体对象,并为对象间的访问施加强制访问控制;安全策略管理器负责管理 FMAC 框架下的多个安全策略,并综合多个安全策略的访问控制决策结果。我们先给出 FMAC 的对象管理器与安全策略管理器的一般模型,再针对 Unix 类操作系统讨论了它们

*)本文的研究得到了国家 863 重大软件专项"服务器操作系统内核"(项目编号:2002AAIZ2101)资助。**易晓东**　博士生,研究方向:操作系统安全与形式化方法;**杨学军**　教授,博导,研究方向:并行与分布处理、信息安全、操作系统、软件工程。

的具体实现方法。

本文的贡献主要有：一是提出了统一的安全策略模型，基于该安全策略模型可以方便快捷地定制出轻量级的访问控制类和信息流类的安全策略；二是提出了在操作系统中支持轻量级安全策略的FMAC框架的理论模型；三是针对开发安全操作系统常用的Unix类操作系统给出了FMAC框架的实现方法，并基于TrustedBSD操作系统实现了一个FMAC框架原型。

本文第2节比较了相关工作，主要是与FLASK进行了比较。第3节给出了FMAC模型，包括基于标记迁移系统LTS的安全策略模型、对象管理器模型和安全服务器模型。第4节介绍了FMAC框架在Unix类操作系统中的实现，并给出一个基于TrustedBSD的实现原型，最后小结了全文。

2 相关工作

与FMAC最相关的工作是FLASK，它是一个通用的强制访问控制框架，由对象管理器和安全服务器两部分组成，对象管理器对操作系统对象进行标记，施加安全策略的访问控制决策并对这些决策进行缓冲以提高效率，还处理策略的动态变化对对象的影响。安全服务器的功能是管理多个安全策略与安全策略的动态变化，对访问控制进行决策。FLASK的安全策略及其实施机制是分离的，从而提供了高度的灵活性。另外，FLASK实现了强制访问控制，因此大大提高了操作系统的安全性。FMAC借鉴了FLASK结构的安全策略及其实施机制分离的思想，从而也具有FLASK的优点。

为了规范和简化安全策略的定义与实现，FMAC定义了基于标记迁移系统LTS的安全策略模型，基于该策略模型可以方便地定制访问控制类和信息流类的安全策略，并能为安全策略定义时间、施加对象等施加条件，从而满足了当前的安全需求。FLASK没有定义安全策略模型，也没有提供对方便灵活地定义安全策略的支持。FLASK框架下的安全模型有的采用编码的形式实现为内核模块(如MLS策略等)，有的则需要多个复杂的配置文件(如TE策略等)。

FLASK的TE策略被认为具有高度的灵活性，但相比本文提出的安全策略模型，TE策略的类型转换只在执行一个程序时才进行，而在我们的安全策略模型中，任何访问操作都可能导致访问控制状态的转换，这可以进一步提高灵活性，并具有更好的最小特权安全性质。另外，FLASK的TE策略实现了系统所有的安全需求，导致TE策略的配置相当复杂。而FMAC可以根据不同的安全需求，选择该安全需求所涉及到的主客体及它们之间的访问操作，定制安全策略，从而简化了系统的安全配置。例如，我们针对文[3]中的六个安全需求可以使用FMAC的六个不同的安全策略来实现。

FMAC与FLASK相似，也是由对象管理器和安全策略管理器组成的，这是因为这种结构易于实现安全策略与实施机制的分离。但是，FMAC模型的对象管理器更进一步描述了对象的组织形式(例如在Unix类操作系统中以层次方式组织)，从而便于可视化和方便快捷地确定安全策略所约束的实体和定制安全策略；FMAC模型的安全策略管理器引入了安全策略的施加条件，为安全策略的施加提供了灵活性，从而在不影响安全性的基础上提高了系统的易用性。

与本文工作紧密相关的还有TrustedBSD操作系统的安全框架MAC[4,5]，它也是一个强制访问控制框架，其安全策略基于框架提供的访问控制钩子函数实现为TrustedBSD的KLD模块，从而可以动态加载与卸载。MAC框架的设计思想与FLASK比较相似，因此也具有FLASK的缺点。

3 FMAC模型

FMAC的模型包含两部分：一是FMAC所使用的基于LTS的安全策略模型，二是组成FMAC的对象管理器与安全策略管理器模型。

3.1 基于LTS的安全策略模型

FMAC的安全策略SP定义为一个标记迁移系统LTS(Labeled Transition System)，SP的每个状态对应了主客体之间的一个访问控制关系，我们使用下文中的访问控制矩阵ACM描述。*SP*定义的状态迁移描述了一个访问控制决策请求ACDR导致的访问控制关系的改变。形式化地，FMAC的安全策略定义为$SP=\langle S, init, \Sigma, T\rangle$，其中：

- S为有穷状态集合，每个状态为一个访问控制矩阵ACM；
- $init$为初始访问控制矩阵；
- Σ是有穷的访问控制决策请求ACDR(Access Control Decision Request)的集合，任何一次访问控制决策请求都可以导致安全策略模型的状态迁移；
- $T \subseteq S \times \Sigma \times S$，为状态转换关系。

由于安全策略模型的状态被定义为访问控制矩阵，因此FMAC是一个访问控制类的安全策略。这是出于两方面考虑的结果：一是由于访问控制是操作系统安全的最底层实现机制，因此基于访问控制定义的安全策略便于在操作系统中实现；二是从用户的角度来看，用户习惯于使用访问控制思想来定制安全策略，因此基于访问控制的安全策略便于用户定制。

对安全策略*SP*的每个状态，我们使用访问控制矩阵ACM(Access Control Matrix)来描述主客体之间的访问控制关系。ACM在访问矩阵AM(Access Matrix)[6]的基础上修改得到，形式化地定义为$ACM=\langle SUBJ, OBJ, ACT, f_{AC}\rangle$，其中：

- $SUBJ$与OBJ为有限的主客体集合；
- ACT为有限的主客体之间的访问操作集合；
- $f_{AC}: SUBJ \times OBJ \times ACT \rightarrow \{grant, deny, unknown\}$为主客体之间的访问控制函数。

由于访问控制类安全策略必须明确给定任意主客体间的任意操作是许可还是禁止，而这个工作是一件极其麻烦的事情[7]，因此我们为每个状态对应的访问控制函数的值域中增加了一个值unknown，以描述安全策略没有明确规定该主客体对应的操作的权限。一般用户定义安全策略时，只定义了部分感兴趣的主客体及它们之间操作的权限，则其它情况对应的权限全部为unknown。如果某安全策略返回unknown，则根据访问控制决策请求所涉及到的主体、客体与访问操作，可以将unknown细分为主体未知、客体未知、操作未知及它们的组合共七种情况。在安全策略定义时，用户可以使用缺省值法[8]确定各种情况下unknown的取值(grant或deny)，也可以将unknown值传递给下文的安全服务器，由它根据实际情况确定取值。

FMAC的安全策略模型是基于访问控制机制定义的，但它不仅能定义出访问控制类的安全策略，而且能定义出信息流安全策略，如多级安全MLS(Multi-Level Security)策略、中国墙(Chinese-Wall)策略等。另外，用户可以充分利用LTS

模型的状态迁移能力定义出高度灵活和复杂的信息流安全策略。

3.2 FMAC 的对象管理器

对象管理器负责对操作系统中主客体的管理，操作系统中的主客体是动态创建和销毁的，因此理论上它们的数量是无限的。但安全策略模型只能处理有限数量的主客体，因此对象管理器的任务是将数量无限且动态变化的操作系统主客体映射到一个有限且固定的集合。

设操作系统的主体集合为 SUBJECT，客体集合为 OBJECT。为了增强灵活性，我们允许 SUBJECT∩OBJECT≠ϕ，即允许一些操作系统中的实体既是主体又是客体。对象管理器的一个对象映射定义为 $OM=\langle LABEL,\pi_s,\pi_o\rangle$，其中：

- $LABEL$ 为一个有穷的安全标记集合；
- $\pi_s: SUBJECT\rightarrow LABEL$，$\pi_o: OBJECT\rightarrow LABEL$，$\pi_s$ 与 π_o 为对象映射函数，分别将主体与客体映射到 LABEL 集合，即为系统中的每个主客体都分配一个安全标记。由于 SUBJECT∩OBJECT≠ϕ，故操作系统中的同一实体可能被 π_s 与 π_o 各分配一个安全标记。

由于集合 SUBJECT 与 OBJECT 都是动态变化的，因此映射函数 π_s 与 π_o 是不确定的。为了定义出确定的映射函数，我们引入属性的概念。根据系统的特征，为主客体各指定一个确定的属性向量，不同的主客体对应于各自属性向量的不同取值。形式化地，主客体的属性向量分别定义为：

主体属性：$\overline{\eta_s}=\langle\eta_s^0,\eta_s^1,\cdots,\eta_s^m\rangle$，其中 $\eta_s^i\in D_s^i(0\leqslant i\leqslant m)$，$D_s^i$ 为 η_s^i 的有穷定义域；

客体属性：$\overline{\eta_o}=\langle\eta_o^0,\eta_o^1,\cdots,\eta_o^n\rangle$，其中 $\eta_o^i\in D_o^i(0\leqslant i\leqslant n)$，$D_o^i$ 为 η_o^i 的有穷定义域。

此时 π_s 与 π_o 可以确定地定义为 $\pi_s:\{\overline{\eta_s}\}\rightarrow$LABEL，$\pi_o:\{\overline{\eta_o}\}\rightarrow$LABEL，其中 $\{\overline{\eta_s}\}$ 与 $\{\overline{\eta_o}\}$ 分别表示向量 $\overline{\eta_s}$ 与向量 $\overline{\eta_o}$ 的所有可能取值的集合。

将操作系统中的主客体映射到安全标记集合 LABEL 后，我们就可以在 LABEL 集合上定义安全策略的访问控制矩阵 $ACM=\langle SUBJ,OBJ,ACT,f_{AC}\rangle$，其中 $SUBJ\subseteq LABEL$，$OBJ\subseteq LABEL$。

FMAC 的对象管理器只定义了主客体的属性，没有限定以何种方式组织主客体，因此在实现 FMAC 时可以灵活地根据具体操作系统的特点组织主客体。例如，可以使用用户—组或基于角色的方式组织主体，从而降低管理代价。

3.3 FMAC 的安全策略管理器

安全策略管理器负责对安全策略进行管理，并根据安全策略对访问控制进行决策。对操作系统的一个访问控制决策请求 ACDR，先由对象管理器根据映射函数将之转换为安全标记之间的访问关系，再由安全策略管理器调用安全策略进行决策。

FMAC 的每个安全策略都定义了一个施加条件 EC(Enforcement Condition)，用于描述安全策略的应用场合，从而实现了安全策略灵活地加载与卸载。安全策略的施加条件可以是施加时间、施加对象等，具体的施加条件由操作系统实现 FMAC 时具体定义。其中施加对象可以定义为施加主体，如某安全策略只施加于某用户；可以定义为施加客体，如某安全策略只施加于某文件；也可以定义为施加操作，如某安全策略只对读操作起作用，同时可以定义为它们的组合。施加条件 EC 由安全服务器检查，对一次访问控制决策请求 ACDR，安全策略管理器只调用符合施加条件 EC 的安全策略对此次 ACDR 进行决策。

安全策略管理器必须综合多个符合施加条件 EC 的安全策略的决策结果，并做出一个确定的访问控制决策(grant 或 deny)。为此，安全策略管理器必须进行两方面的工作：一是处理单个安全策略返回 unknown 的情况，二是以一种合理的方式综合多个安全策略的决策结果。

一般情况下，用户在定义安全策略时就已经使用缺省值法定义了所有情况下 unknown 的取值，否则 FMAC 根据安全访问上下文(如访问的历史信息、安全性要求高低、访问主体的信誉等)决定取值为 grant 还是 deny。

FMAC 并不限定以何种方式综合多个安全策略的访问决策结果。在具体的操作系统中实现 FMAC 时，可以采用文[7]中的并行组合(Parallel Composition)、顺序组合(Sequence Composition)、条件组合(Conditional Composition)等多种组合方式，但一般的操作系统都会采用并行组合方式方式，对安全策略 $SP_1,SP_2,\cdots,SP_n$ 而言，其效果等价于 $SP_1\wedge SP_2\wedge\cdots\wedge SP_n$，即只有满足所有的安全策略，访问才被许可。

安全策略管理器的另一个重要功能是安全策略管理功能，包括添加一个新策略到系统、从系统中卸载一个正在使用的策略等，是实现多策略和策略灵活的基础。FMAC 的安全策略管理功能与 FLASK 基本相同。

4 FMAC 在 Unix 类操作系统中的实现

考虑到当前开发安全操作系统的主流方法是基于 Unix 类操作系统进行安全增强，因此我们讨论 FMAC 在 Unix 类操作系统中的实现。我们充分利用 Unix 类操作系统内核提供的访问控制机制，如 Linux 的 LSM(Linux Security Modules)机制、TrustedBSD 的 MAC 框架等，来实现 FMAC 所需的底层强制访问控制。

4.1 对象管理器的实现

Unix 类操作系统中，FMAC 的主体只有进程，其属性向量 $\overline{\eta_s}$ 由三个基本属性和多个策略属性组成。三个基本属性为进程所属的用户、进程对应的应用程序文件和应用的父进程，进程的策略属性是系统的安全策略赋予该进程的安全标记。与 SELinux、TrustedBSD 等操作系统一样，系统中的每个安全策略都有一个可以自定义数据结构的策略属性。我们使用基于角色定权模型 RBA(Role Based Authorization)[1] 对主体进行组织，RBA 基于 $RBAC_3$ 模型[9] 实现，利用角色的继承与约束大大减小了安全管理的复杂程度。

FMAC 的客体定义为一个分类的层次结构，客体的属性向量 $\overline{\eta_o}$ 由分类层次标识和策略属性两部分组成，其中策略属性由所有安全策略自定义数据结构的策略属性组成。文[4，10]分别给出了 Linux 与 TrustedBSD 操作系统的强制访问控制框架所控制的客体，包括进程、打开的文件描述符对象、文件系统对象、文件与管道对象、目录对象、网络套接字对象、TCP 与 Unix 流套接字对象、网络接口和端点对象等，每类对象对应不同的操作。FMAC 的对象管理器对操作系统中的所有客体进行了层次式归类，并知道新创建的客体属于哪一子类，从而可以控制客体的类别与对应的操作。FMAC 的层次式归类将操作系统的客体组织成层次式结构，例如对于 Linux 的文件系统，其层次结构如图 1 所示，其中方框表示层次式客体的类别，圆框表示客体的操作。由于 Linux 操作系统中进程是唯一的主体，因此我们将主客体之间的操作作为客体的属性直接与客体绑定，从而既方便安全策略的定义，又

便于客体管理器管理系统客体及其有效操作。

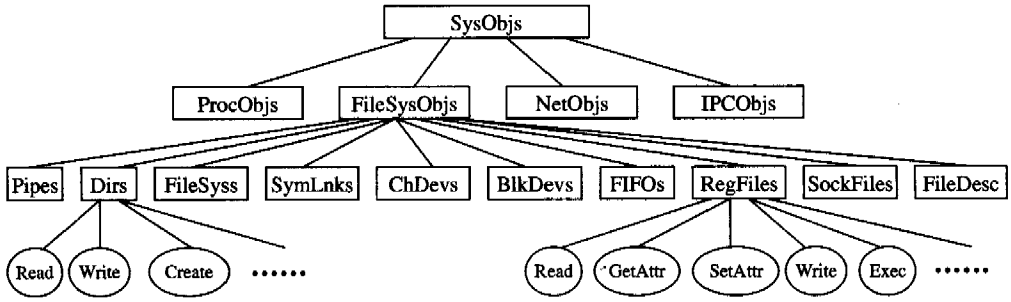

图 1 Linux 文件系统客体层次分类示意图

在 Unix 类操作系统中实现 FMAC 时，安全标记集合 LABEL 中的每个安全标记都被定义为一个整数，类似 FLASK 的安全标识符 SID。根据定义，对象映射函数 π_s 与 π_o 分别将系统的主客体根据属性映射到 LABEL 集合，在 FLASK 中该映射关系是通过策略编译器完成的，而我们在系统初始化时根据安全策略的访问控制关系动态确定，相当于在系统初始化时动态编译安全策略，从而能更好地适应策略的变化。

4.2 安全服务器的实现

FMAC 的安全策略作为内核模块实现，因此 FMAC 的安全服务器与内核模块管理器协同工作，完成对安全策略模块的加载、卸载等管理。

FMAC 的安全服务器目前支持两种安全策略施加条件：一种是基于时间的施加条件，另一种是基于施加对象的施加条件，详见下一小节对安全策略定义的讨论。

对于 unknown 值，FMAC 的安全服务器总是将其转换为 deny，以获得较高的安全性。另外，FMAC 采用并行组合方式，即只有满足所有的安全策略，访问才被许可。

4.3 安全策略的定制

FMAC 除了使用标记迁移系统 LTS 规范安全策略的定制外，还提供了可视化的机制，辅助用户定制安全策略本身与安全策略的施加条件。定制一个安全策略包括三个方面的工作：一是选定安全策略所约束的主客体及它们之间的操作；二是确定安全策略的状态转换事件，并对每个状态下的主客体之间的操作赋值(grant 或 deny)；三是确定各个状态的 unknown 在各种情况下的取值。对安全策略的施加条件而言，时间类的施加条件设定非常直观，施加对象的选定与安全策略定制时主客体及其操作的选定是相同的。

主客体及操作的可视化选择由对象管理器提供支持。由于 FMAC 的对象管理器将主体组织为基于角色的结构，将客体按层次分类，并将各客体对应的操作附加于客体的子类上，因此主客体及其操作的选择都非常直观。例如在图 1 中，我们定制安全策略时可以使用 SysObjs. FileSysObjs. RegFiles 代表系统中的所有常规文件，而 SysObjs. FileSysObjs. RegFiles. xxx_file 具体指定 xxx_file 文件。SysObjs. FileSysObjs. Dirs. Create 表示在系统的所有目录中创建的操作，可以进一步把创建划分为创建目录、创建文件等操作。

在操作系统中，有些操作可能会涉及到多个客体。例如装载一个文件系统时，要求主体对文件系统的设备文件及该文件系统的装载点目录有 Search 的权限，同时要求主体具有对该文件系统装载的权限及装载点目录装载文件系统的权限，另外还要求被装载的文件系统的根目录与装载点目录之间有装载关联权限[10]。为此，FMAC 的对象管理器还管理一个操作对应的多个主客体之间的权限关联关系，既使得系统可以全面检查一个操作对应的所有访问控制，也可以帮助用户完整地定义每个安全操作对应的所有访问控制权限。

安全策略状态转换事件被定义为访问控制决策请求 ACDR，该 ACDR 的主客体及其操作同样可以使用对象管理器提供的支持可视化地选定。另外，对七种情况下的 unknown 取值的确定也是非常直观的。

4.4 FMAC 的实现原型

我们基于 TrustedBSD 的强制访问控制框架 MAC 实现了 FMAC 的一个原型。该 FMAC 原型只实现了对 TrustedBSD 的常规文件系统的控制，即只考虑常规文件和目录两类客体。下面我们先介绍 TrustedBSD 的强制访问控制框架，再介绍我们的实现方法。

TrustedBSD 的 MAC 框架已经实现了对象管理与标记功能，并能够对文件系统的标记使用 UFS2 文件系统的扩展文件属性进行持久存储。TrustedBSD 的安全策略以内核模块的形式实现，MAC 框架提供了接口，供每个安全策略内核模块为所有的内核对象分配一个自定义结构的安全标记。每次内核访问时，MAC 都会把该访问所涉及到的内核对象对应各策略的安全标记交给各策略模块进行访问决策。

我们将 FMAC 框架实现为 MAC 的一个策略模块，并以安全配置文件的形式在 FMAC 的基础上定义多个安全策略。下面我们扼要介绍 FMAC 的对象管理器与安全服务器的实现。

FMAC 的对象管理器为主体(进程)定义了四个基本属性，分别是该进程的用户、该进程的应用程序文件标记、该进程的父进程 ID 与该进程的角色。FMAC 的客体(我们只考虑常规文件和目录)已经被打上了标记，FMAC 维护一个表，记录所有文件与目录标记的层次关系，以及各个文件与目录对应的操作。对于每个操作，FMAC 记录了该操作所涉及的所有主客体以及所需的访问控制权限。

FMAC 的每个策略对应于一个配置文件，系统启动时读入并由 FMAC 进行处理。对安全策略的每个状态，FMAC 的安全服务器都生成一张访问控制规则表，记录该状态的访问控制权限。对于安全策略每个状态中的每个访问控制语句，FMAC 的安全服务器为该语句涉及的主体与客体分别分配一个整数，调用对象管理器的接口使用这两个整数分别标记主客体，最后在安全服务器中记录这两个整数的访问控制关系。对象管理器标记客体时，会将该客体在客体层次关系树中的所有子客体都进行标记。如果在处理安全策略的某个访问控制语句时，某个对象已经被标记过，则为其新分配一个标

记，并通知安全服务器增加两类访问控制规则，表示该客体适用于两类访问控制规则。例如，某访问控制语句规定某主体可以读某一类文件，经过处理，我们假设安全服务器为主体分配整数标记为1，为客体分配整数标记为2，则安全服务器记录了标记为1的主体对标记为2的客体有读的权限。此时，如果该客体代表的某个文件已经被分配了标记9，则该客体被重新分配一个标记，假设为11。安全服务器在已有的访问控制规则中搜索包含标记9的访问控制规则，每找到一条，就生成一条新规则，以11替换9。另外，安全服务器再添加一条规则，记录标记为1的主体对标记为11的客体有读的权限。

FMAC实现了五个施加于全系统的安全策略，分别是满足文[3]中保护内核完整性及保护系统文件完整性要求的两个安全策略、基于角色的访问控制策略、能力策略和多级安全策略，这些策略都只对文件和目录进行约束。FMAC还实现了一个示例性的施加于某个普通用户的安全策略，该安全策略规定所有该用户的程序都不能访问其Home目录下的某个子目录。

对于一个访问控制决策请求（由MAC框架调用FMAC的钩子函数），FMAC的安全服务器可以直接判断其施加条件是否满足，并使用并行组合方式综合多个安全策略的决策结果。

小结 针对当前操作系统的安全需求，本文在FLASK安全体系结构的基础上提出了一种灵活的强制访问控制框架FMAC，FMAC模型由基于LTS的安全策略模型、对象管理器模型和安全服务器模型组成。本文论述了Unix类操作系统中FMAC的实现，并基于TrustedBSD实现了一个FMAC原型。

接下来的工作主要是完善FMAC模型和实现，并采用访问控制决策缓冲等机制解决实现原型中暴露的对操作系统性能的影响问题。

参考文献

1 易晓东，何连跃，杨学军. 安全操作系统基于角色的授权机制. 计算机工程与科学，2004(增刊)

2 Spencer R, Smalley S, Loscocco P, et al. The Flask Security Architecture: System Support for Diverse Security Policies. In: Proc. of the Eighth USENIX Security Symposium, 1999. 123～139

3 Loscocco P, Smalley S D. Meeting Critical Security Objectives with Security-Enhanced Linux. In: Proceedings of the 2001 Ottawa Linux Symposium, 2001

4 Watson R, Feldman B, Migus A, et al. Design and Implementation of the TrustedBSD MAC Framework. In: Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX III), 2003

5 Watson R, Morrison W, Vance C, et al. The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0. In: Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference (FREENIX '03), 2003

6 Lampson B W. Protection. In: 5th Princeton Symposium on Information Science and Systems, 1971

7 Siewe F, Cau A, Zedan H. A compositional framework for access control policies enforcement. FMSE'03, ACM, 2003

8 Hale R W S. Programming in Temporal Logic: [PhD thesis]. Trinity College, University of Cambridge, 1988

9 Sandhu R S, Coyne E J, Feinstein H L, et al. Role-Based Access Control Models. IEEE Computer, 1996, 29(2): 38～47

10 Loscocco P, Smalley S. Integrating Flexible Support for Security Policies into the Linux Operating System. In: Proc. of the FREENIX Track of the 2001 USENIX Annual Technical Conf. 2001

（上接第143页）

然后，对子目标的证明应用推理规则，从而产生原始目标的证明。

这样的证明策略可以表示为一个函数，将一个证明目标映射为一个包含子目标列表和一个确认证明的序偶。一个确认证明可将子目标的证明映射为原始目标的证明，即

$$type\ tactic = goal \rightarrow (goal\ list \times (proof\ list \rightarrow proof))$$

函数 *tactic* 是证明系统中自动应用规则的一个基本工具，它检查给定目标，并将其分解成一些要解决的子目标，向前或向后推导出新的证明公式或确定一个证明是否是该公式的证明。*Tactic* 的主要用途是将一个证明转换成自然推理。例如，将函数应用到目标 g 上返回$([],f)$，则 $f[]$就是 g 的证明。

定理证明器通常使用交互式的目标制导策略，用户可以指定一些推理步骤，证明的细节是自动进行的。由此，用户可以组合所谓的原始证明，它们可以是推理规则，或是判定过程的调用。例如，PVS有许多原始证明和一个小的 *tactic* 语言，而Isabelle/HOL系统有一个完整的程序设计语言(ML)，作为 *tactic* 语言来组合复杂的原始证明。

如果希望通过选择、复合或重复的方法组合原始的 *tactic*，则可以构建组合子 *tacticals*，一些基本的组合子如：

T_1 then T_2　应用 T_1，然后应用 T_2；

T_1 orelse T_2　试应用 T_1，若失败，则再应用 T_2；

repeat T_1　重复应用 T_1，直至失败。

结论 很多广为使用的证明器都是基于某一类型理论的，这表明：

- 类型对于组织形式化的知识是有用的。
- 对象的类型可以将一些有用的信息传送给证明者。
- Curry-Howard同构给出了一种简便的方法，可以从证明中进行程序综合。

应该看到，目前形式化的正确性证明还是一项十分复杂和耗时的工作。证明简单的程序是十分容易的，一旦命题十分复杂，则证明的复杂度显著增长。当然，培训和利用工具软件有助于缩短所需时间。

形式化证明的一个优点也是明显的，至少可以使我们更为深刻地认识程序的规律，更好地理解算法，从而改进并简化代码及文档。

参考文献

1 Apt K R, Olderog E R. Verification of Sequential and Concurrent Programs Springer-Verlag, 1997

2 Csornyei Z. Type Systems, Lecture Notes (2003). http://people.inf.elte.hu/csz (In Hungarian)

3 Dunfield J, Pfenning F. Tridirectional Typechecking. In: POPL'04, 2004

4 Harper R, Pfenning F. Type Refinements, Project Description, 2001. http://www-2.cs.cmu.edu/1triple/triple.pdf

5 Pierce B C. Types and Programming Languages. The MIT Press, 2002

6 Schwartzbach M I. Polymorphic Type Inference BRICS Lecture Series, LS-95-3, 1995

7 Sørensen M H B, Urzyczyn P. Lectures on Curry-Howard Isomorphism. Lecture Notes, University of Copenhagen, University of Warsaw, 1999

8 Zoltan C. Type Systems and Program Verification. In: 6th International Conference on Applied Informatics, 2004, 27～31