基于多安全机制的 Linux 应用沙箱的设计与实现

李 晨[1] 涂碧波[2] 孟 丹[2] 冯圣中[1]

[1](中国科学院深圳先进技术研究院 深圳 518055)
[2](中国科学院信息工程研究所 北京 100093)

摘 要 文章设计了一个具有自己独立工作目录的 Linux 应用沙箱，可为用户对不信任的应用程序提供一个独立和安全的运行环境，应用程序在沙箱中所做的操作对主机不会造成任何影响。该沙箱提供了文件系统隔离、系统资源隔离、物理资源隔离、权能限制和强制访问控制(Mandatory Access Control，MAC)等策略，添加了地址随机化、不可执行页保护等内存保护安全策略。与已有沙箱对比，文章设计的沙箱增加了多种安全机制，提高了系统的安全性，保护了系统的数据安全和用户的个人隐私等。

关键词 Linux 应用沙箱；资源隔离；强制访问控制；权能；内存安全保护
中图分类号 TP 309.2 **文献标志码** A

Design and Implementation of Linux Application Sandbox Based on Multiple Security Mechanisms

LI Chen[1] TU Bibo[2] MENG Dan[2] FENG Shengzhong[1]

[1](*Shenzhen Institutes of Advanced Technology*, *Chinese Academy of Sciences*, *Shenzhen* 518055, *China*)
[2](*Institute of Information Engineering*, *Chinese Academy of Sciences*, *Beijing* 100093, *China*)

Abstract Linux application sandbox is designed for providing an independent, secure operating environment for untrusted applications. The sandbox has its own independent working directory, and the operation of applications in the sandbox has no impact on the host. The sandbox provides filesystem isolation, system resources isolation, physical resources isolation, capabilities limits and mandatory access control (MAC) policies, adding memory protection policies like address randomization and non-executable memory page protection. The sandbox increases several security mechanisms relative to existing sandboxes, improving the system security and protecting the system and user's personal privacy.

Keywords Linux application sandbox; resources isolation; MAC; capabilities; memory protection

收稿日期：2013-10-6
基金项目：国家高技术研究发展计划(863 计划)(2012AA01A401)。
作者简介：李晨，硕士研究生，研究方向为系统软件安全；涂碧波(通讯作者)，博士，副研究员，研究方向为系统软件安全，E-mail：tubibo@iie.ac.cn；孟丹，博士，研究员，研究方向为计算机体系结构、大数据和系统安全；冯圣中，博士，研究员、博导，研究方向为高性能计算、网格计算和生物信息学。

1 引 言

众所周知，大多数病毒会利用应用程序这条路径进行传播。用户在使用计算机时，对一些应用程序的非法操作往往会导致严重的安全问题，给用户带来极大的困扰。例如用户使用的应用程序需要访问网络时，有可能会从不安全的站点下载恶意程序，执行一些非法操作，如盗取用户的敏感信息，干扰用户的日常工作、数据安全和个人隐私等[1]。一种有效的保护计算机安全的方法就是为应用程序提供一个隔离的运行空间，而这可以由沙箱来提供[2-4]。本文基于多种资源隔离和安全机制设计并实现了一种应用在 Linux 操作系统上的沙箱。相对于现有的沙箱技术而言，该沙箱简化了实现机制，且提供了更高的安全性和隔离性。

2 沙箱的设计与实现

2.1 现有沙箱技术分析

现有的沙箱技术有很多种，如 FreeBSD Jails 是 FreeBSD 平台上一种基于容器的虚拟化技术，它扩展了传统的 Chroot 机制，为进程提供了一个具有独立的文件系统、进程和网络空间的环境。当进程调用了 jail 系统调用后，它就被限定在这个 jail 内，且不能脱离。该机制的缺点是不能够隔离物理资源，且不具备安全机制。Seccomp 是 Linux-2.6.23 版本之后支持的一种沙箱机制，该机制为进程提供了一种“安全”模式。在这种模式下，只允许沙箱内的进程使用指定的 4 种系统调用：exit()、sigreturn()、read()和 write()[5]，否则进程将会被终止，但该机制的实现需要修改内核及应用程序，且只支持“纯计算型”代码的应用程序，安全机制单一。BufferZone 是通过在用户电脑上指定的目录中创建一个虚拟区域，当该沙箱在运行程序时，程序下载的所有文件都将存储在这个虚拟区中，进而使它们与操作系统的其他部分隔离开来，因此恶意软件只能对虚拟区造成影响。但这种机制的开销大，且虚拟机本身存在很多安全问题。DefenseWall 将运行程序分为两类：系统自带的文件程序作为可信任程序，而第三方软件程序作为非信任程序。它还可随时对非信任程序进行跟踪监听。虽然该机制内置一些策略和规则来限制不可信程序的运行，但它不具备网络过滤的功能，且不能实现资源隔离。SELinux 是一种强制访问控制机制，它首先给系统的主体和客体打上不同的安全标记，然后通过定制访问控制规则来限制主体对客体的访问，但这种机制的配置比较复杂，易用性较差。表 1 为一部分典型的沙箱的特性对比。

综上所述，现有的沙箱中有一部分需要修改内核和应用程序本身，易用性差，且使用的隔离和安全策略有限，不能够很好地防范各种恶意攻击[6,7]。

本文设计的沙箱可以为不可信应用程序提供一个系统资源、物理资源和文件系统隔离的运行环境，同时还添加了强制访问控制策略、权能限制和内存保护功能，在防范应用程序漏洞攻击的同时还可以防止部分系统漏洞攻击，而且无需修改内核及应用程序本身。与已有的沙箱对比，本文设计的沙箱提高了系统的隔离性和安全性，且对性能造成的影响在可控的范围之内。

2.2 沙箱的设计目标

为了保证沙箱的隔离性和安全性，本文为沙箱提出了以下两个设计目标：

(1) 实现资源隔离

① 沙箱拥有自己独立的文件系统；

② 实现沙箱间及沙箱与主机间的系统资源和物理资源隔离。

(2) 实现多种安全机制

① 使用强制访问控制策略实现主机的关键文件和目录不受沙箱影响；

表 1　典型沙箱技术对比

Table 1.　Typical sandbox technology contrast

名称	原理	隔离性	安全性	易用性	性能开销
FreeBSD Jails	对 Unix 传统的 chroot 机制的一种扩展。通过 jail 系统调用来供用户使用的，进程及其子进程被置于 Jail 之内	提供了隔离的文件系统，隔离进程、进程间通信及网络资源（特定的 IP 地址），隔离性较强	弱	需要修改内核代码，易用性较差	小
Seccomp	只允许程序使用 exit()、sigreturn()、read() 和 write() 四种系统调用	只能够限制应用能够使用的系统调用，且不能动态分配内存、不能使用新的文件描述符等，局限性比较大	弱	需要修改内核代码，易用性较差	小
BufferZone	建立一个与系统一样的虚拟系统，将不可信的应用程序安装在这个系统上	使用虚拟系统，隔离性强	强	强	大
DefenseWall	它将程序分为可信任和不信任，不信任程序会与信任程序隔离，并且会受到很多内置策略	限制对本地文件的读写，但是没有网络过滤功能，隔离性较弱	弱	较强	中
Sandboxie	程序运行时，程序会通过沙箱来读取数据，然后最后写入沙箱虚拟出来的文件中，不会写入硬盘	在应用程序和硬盘之间进行隔离，隔离效果一般	强	强	中

② 限制沙箱可以使用的权能(Capabilities)；

③ 为沙箱添加内存保护安全策略。

2.3　沙箱的实现

2.3.1 资源隔离

首先，利用命名空间(Namespace)机制实现沙箱内系统资源的隔离。所有的系统资源都默认为全局管理，Linux 系统提供了 PID、IPC 和 Internet 等多个 Namespace，而且每个 Namespace 的资源相对于其他的 Namespace 均为透明。我们为每个沙箱建立一个独立的 Namespace，这样网络、PID 和 IPC 等资源就属于特定的 Namespace。这种机制的实现依靠在启动沙箱进程后调用 clone() 系统调用时，指定相应的 flags，flags 标识的是我们要为沙箱建立的新 Namespace 类型。我们将 PID Namespace 和 IPC Namespace 一起使用可以使沙箱中的进程彼此不可见、不可通信，实现了进程间的隔离；挂载 Namespace 和网络 Namespace 一起使用可以为沙箱虚拟出一个具有独立主机名和网络空间的环境，对于应用程序而言，沙箱就像网络上一台独立的主机一样。

其次，隔离沙箱可以使用的物理资源。沙箱利用控制组(Control Groups，Cgroups)子系统来实现这一功能。Cgroups 是一种可以隔离、限

制和记录进程组所使用的物理资源(如 CPU、memory 和 IO 等)的机制。例如，内存子系统用来限制进程组可以使用的内存上限，一旦进程组使用的内存达到了限额以后还继续申请内存，系统就会终止该进程并报错。我们将沙箱中的进程作为一个进程组来进行管理，这样就可以避免沙箱中的恶意程序想要霸占某些物理资源的行为。

此外，我们还为沙箱定制了一个完全独立的文件系统，如图 1 所示。

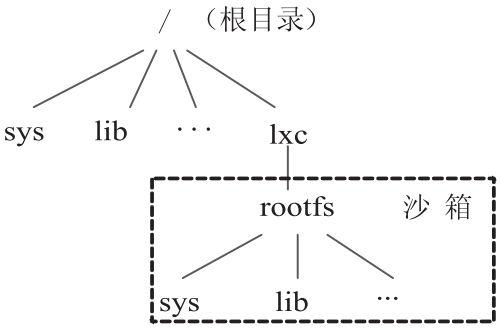


图 1 沙箱的独立文件系统

Fig. 1. Rootfs of Sandbox

该文件系统相当于主机文件系统的一个简化副本，其中包含了应用程序本身以及它执行时需要的配置文件、动态链接库和特殊设备等，沙箱中的进程只能在这个文件系统中进行操作。在沙箱的文件系统中，主机系统中默认禁止访问的敏感目录文件只是一个拷贝，所以无论恶意应用程序在沙箱中做任何操作都不会对主机产生任何影响。该机制的实现利用了系统调用函数 chdir()，在启动沙箱时调用该函数，即可更改当前进程的工作目录。

2.3.2 安全策略

除了以上提供的资源隔离策略，我们还为沙箱引入了一些安全策略。首先是强制访问控制机制，强访机制就是将系统中的进程、文件等对象分为主体和客体两种，并给主、客体添加相应的标签，定制特定的强制访问控制规则来限制主体对客体的操作[8]。我们使用简单的文本标签(如 host)标记主机上的关键文件和目录，给沙箱的独立文件系统以及沙箱进程添加沙箱标签(可以用沙箱名作为标签，如 vs 1、vs 2)，设置沙箱使用的访问控制规则如表 2 所示。其中，第一列为主体标签，第二列为客体标签，第三列为主体可以对客体执行的操作，沙箱对标有 host 标签的对象不能做任何操作，沙箱互相之间也不能做任何操作，这样就可以保证沙箱本身及沙箱中的恶意程序不会对主机的关键文件和目录产生影响。

表 2 沙箱的强制访问控制规则

Table 2. MAC rules of sandboc

主体标签	客体标签	访问
vs 1	host	—
vs 2	host	—
host	vs 2	rwxa
host	vs 1	rwxa
vs 1	vs 2	—
vs 2	vs 1	—

其次，对沙箱中的进程进行权能限制。Linux Kernel 自 2.1 版开始就有了权能的概念，Linux 支持权能的主要目的是细化根用户的特权，利用这一机制，我们可以根据实际的安全需要来控制沙箱拥有的权能范围，从而防止攻击者利用恶意应用程序趁机获取控制系统的特权，对系统安全造成威胁。该机制的实现是通过在沙箱中启动进程时，先获取进程拥有的权能，然后使用 cap_clear_flag()函数来清除禁止该沙箱使用的权能。

此外，为沙箱添加了地址随机化、不可执行页保护等安全策略，这些安全策略可以防止由缓冲区溢出漏洞导致的攻击行为。例如，如果攻击程序熟悉进程的地址空间，从而将程序的执行流程跳转到恶意代码的位置，危害系统安全。而利用地址随机化策略使程序地址随机化，使得攻击

者无法将程序执行流程跳转到预期位置，从而阻止攻击代码执行。这样不仅可以对已知漏洞进行防护，还能对未知漏洞利用攻击进行防御。

3　性能分析及测试

3.1　安全性分析

因为沙箱具有独立的文件系统，所以从沙箱中启动应用程序后，在执行过程中所做的操作，如下载的文件以及在网页上安装的软件都被重定向到沙箱独立的文件系统中，可以随时删除且不会对主机系统造成任何影响；此外，沙箱引入了 Namespace 机制和 Cgroups 机制，实现了沙箱与系统间系统资源和物理资源的隔离；权能机制使应用程序及沙箱都不可能进行除了赋予给它们的权能之外的任何特权操作；强访策略可以限制沙箱程序读取或修改系统的关键文件和敏感数据；地址随机化和不可执行页保护策略防止了恶意代码的缓冲区漏洞攻击行为。如图 2 所示为沙箱使用的各种机制及达到的隔离和安全效果。

3.2　测　试

测试环境：

Intel（R）Core（TM）i5-3470 CPU 3.20 GHz

Fedora 12 linux-2.6.36

（1）安全性测试

测试1：对存在安全漏洞的应用程序直接执行与在沙箱中执行的结果进行测试。根据 Mozilla 官网安全公告，火狐浏览器（Firefox）的某一版本存在高危级别的漏洞 MFSA 2010-65。如果向 document.write（）发送一个超长字符串，可能会导致文字渲染程序出现混乱，堆栈内存的一部分被串数据所覆写。攻击者可利用此漏洞导致用户浏览器崩溃，并可能执行任意代码。

测试过程如下：

① 向 document.write（）发送超长字符串；

② 打开 Debuggy，直接运行 Firefox，在 Debuggy 中多次查看 Call Stack 的内容。

Call Stack 被写入为 41414141，也就意味着在进行正常的浏览器操作之后，32 位指令寄存器（Extended Instruction Pointer，EIP）就会变为 41414141，这个地址是不可执行的，返回地址被覆盖，EIP 指向的一段恶意代码（如盗取系统敏感数据）就会立刻被执行，导致火狐浏览器崩溃。图 3 所示为系统打印的错误信息。

但是，将火狐浏览器在沙箱中启动，执行上述操作之后，恶意代码并不会被执行，原因是我

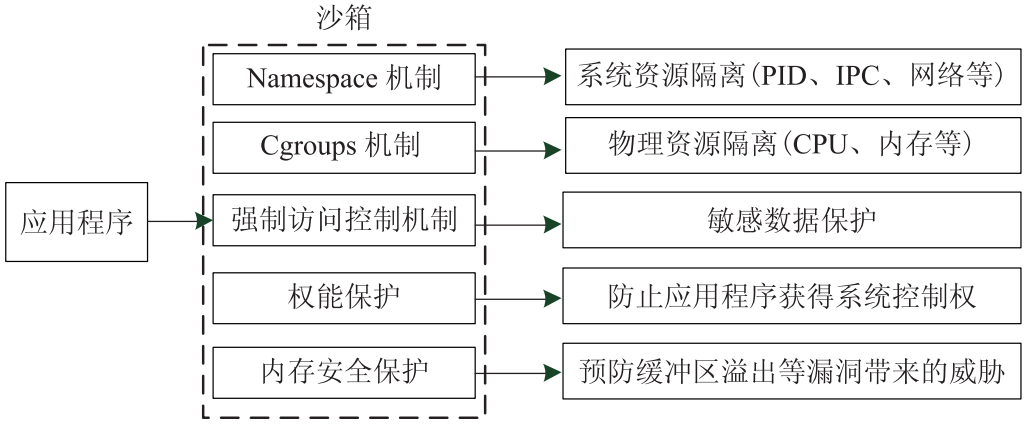


图 2　沙箱使用机制及达到的隔离和安全效果

Fig. 2.　Isolation and security with sandbox mechanisms

们在沙箱中采用了堆栈地址随机化技术。此外，即使火狐浏览器运行时跳转到恶意代码的位置，这段代码也只会在沙箱中运行，它所破坏的仅仅是我们虚拟出来的一个系统副本，并不会对沙箱外的系统造成任何危害，因此，沙箱隔离了存在安全漏洞的应用程序对系统的危害。

```
module <unknown> at 0000:41414141.
Registers:
EAX=00000001 CS=017f EIP=41414141 EFLGS=00010206
EBX=0123d1b0 SS=0187 ESP=041df3b0 EBP=041dfed4
ECX=00000000 DS=0187 ESI=041df3f0 FS=3e6f
EDX=00000000 ES=0187 EDI=00000000 GS=0000
Bytes at CS:EIP:

Stack dump:
41414141 41414141 41414141 41414141
41414141 41414141 41414141 41414141
41414141 41414141 41414141 41414141
```

图 3 系统打印的错误信息

Fig. 3. Error information printed by the system

测试 2：对恶意霸占资源的程序直接执行与在沙箱中执行的结果进行测试。

测试程序关键代码如下：

```
while（1）
 {
   gettimeofday（&tpstart,NULL）;
   while（1）
    {
      gettimeofday（&tpend,NULL）;
      timeuse=1000000*（tpend.tv_sec-
tpstart.tv_sec）+（tpend.tv_usec- tpstart.tv_
usec）;
       timeuse /= 1000;
      char * logmsg = new char［65535］;
     }
    sleep（20 / 1000）;
 }
```

该程序处在 while 循环中，在程序执行时，会循环申请变量，不停地分配内存，导致系统可用内存的数量不断减少。我们测试的结果在经 13.5 s 以后，1 G 的可用内存几乎完全被占用，导致其他应用程序无法执行。之后我们从沙箱中启动该程序，限制沙箱所能使用的内存大小为 100 M。当程序执行时，系统为该程序分配的内存达到限制值以后，应用程序被终止，而其他应用程序仍可正常执行。由此可见，沙箱防止了恶意程序对于资源的恶意霸占，保护了系统的安全。

(2)性能测试

① 应用级性能测试

以三种典型的桌面应用程序 Firefox、Office 和 Kaffeine 为例来分析该沙箱性能损耗。对比直接启动应用程序和从沙箱中启动应用程序的启动时间(以 s 为单位)，结果如图 4 所示。图 4 显示，在沙箱中启动应用程序的时间比直接启动稍慢一些，但是属于可接受的范围。

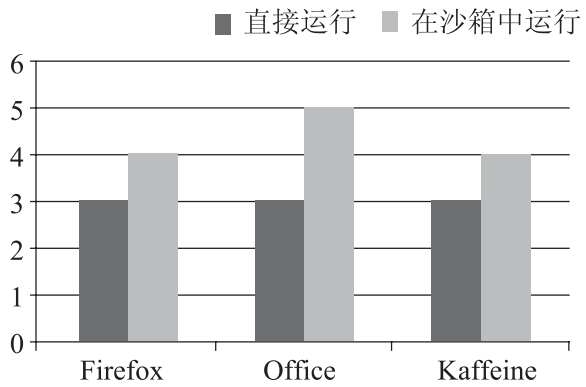


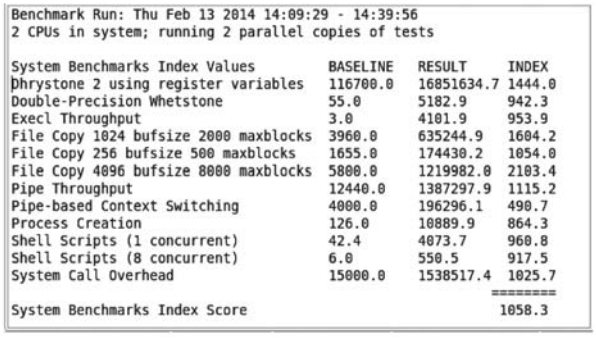
图 4 三种应用程序直接运行和在沙箱中运行时间(s)对比

Fig. 4. Time contrast of three apps running directly and in sandbox

② 系统级性能测试

使用工具 Unixbench 对沙箱及主机进行系统级的性能对比测试。Unixbench 是一个用于评价系统综合性能的开源工具，能够测试包括文档读写、内存操作、进程创建销毁开销和网络等性能开销。正常运行 Unixbench 的测试结果如图 5 所示，而在沙箱中运行 Unixbench 的测试结果如图

6 所示。经对比发现，在沙箱中运行时，系统的性能开销增加了 2.88%，属于可接受范围。

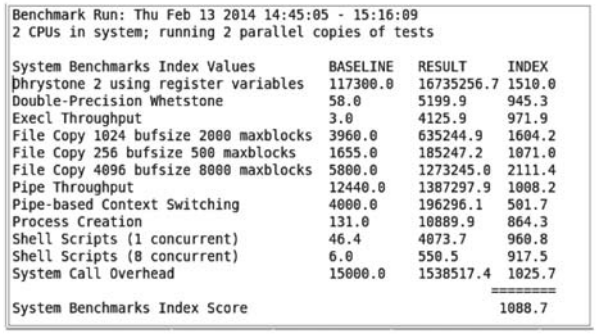


```
Benchmark Run: Thu Feb 13 2014 14:09:29 - 14:39:56
2 CPUs in system; running 2 parallel copies of tests

System Benchmarks Index Values          BASELINE    RESULT      INDEX
Dhrystone 2 using register variables    116700.0    16851634.7  1444.0
Double-Precision Whetstone              55.0        5182.9      942.3
Execl Throughput                        3.0         4101.9      953.9
File Copy 1024 bufsize 2000 maxblocks   3960.0      635244.9    1604.2
File Copy 256 bufsize 500 maxblocks     1655.0      174430.2    1054.0
File Copy 4096 bufsize 8000 maxblocks   5800.0      1219982.0   2103.4
Pipe Throughput                         12440.0     1387297.9   1115.2
Pipe-based Context Switching            4000.0      196296.1    490.7
Process Creation                        126.0       10889.9     864.3
Shell Scripts (1 concurrent)            42.4        4073.7      960.8
Shell Scripts (8 concurrent)            6.0         550.5       917.5
System Call Overhead                    15000.0     1538517.4   1025.7
                                                                ========
System Benchmarks Index Score                                   1058.3
```

图 5　正常运行时的测试结果

Fig. 5.　Test results of normal operation



```
Benchmark Run: Thu Feb 13 2014 14:45:05 - 15:16:09
2 CPUs in system; running 2 parallel copies of tests

System Benchmarks Index Values          BASELINE    RESULT      INDEX
Dhrystone 2 using register variables    117300.0    16735256.7  1510.0
Double-Precision Whetstone              58.0        5199.9      945.3
Execl Throughput                        3.0         4125.9      971.9
File Copy 1024 bufsize 2000 maxblocks   3960.0      635244.9    1604.2
File Copy 256 bufsize 500 maxblocks     1655.0      185247.2    1071.0
File Copy 4096 bufsize 8000 maxblocks   5800.0      1273245.0   2111.4
Pipe Throughput                         12440.0     1387297.9   1008.2
Pipe-based Context Switching            4000.0      196296.1    501.7
Process Creation                        131.0       10889.9     864.3
Shell Scripts (1 concurrent)            46.4        4073.7      960.8
Shell Scripts (8 concurrent)            6.0         550.5       917.5
System Call Overhead                    15000.0     1538517.4   1025.7
                                                                ========
System Benchmarks Index Score                                   1088.7
```

图 6　在沙箱中运行时的测试结果

Fig. 6.　Test results in sandbox

4　总　结

该沙箱利用各种安全策略和隔离机制给不可信任的应用程序提供一个安全隔离的运行环境，配置简单并且无需修改内核和应用程序，具有很强的安全性和隔离性，是一种保护 Linux 系统安全的有效手段。但是，沙箱对系统的性能会产生一些影响，如何进一步降低沙箱性能消耗还是一个亟待解决的问题，这将是我们下一步研究的重点。

参 考 文 献

[1]　程龙, 杨小虎. Linux 系统内核的沙箱模块实现 [J]. 计算机应用, 2004, 24(1): 79-81.

[2]　Acharya A, Raje M. MAPbox: using parameterized behavior classes to confine untrusted applications [C] // Proceedings of the 9th Conference on USENIX Security Symposium, 2000: 1.

[3]　Peterson DS, Bishop M, Pandey R. A flexible containment mechanism for executing untrusted code [C] // Proceedings of the 11th USENIX Security Symposium, 2002: 207-225.

[4]　Provos N. Improving host security with systemcall policies [C] // Proceedings of the 12th Conference on USENIX Security Symposium, 2003: 257-272.

[5]　Bernaschi M, Gabrielli E, Mancini LV. Operating system enhancements to prevent the misuse of system calls [C] // Proceedings of the 7th ACM Conference on Computer and Communications Security, 2000: 174-183.

[6]　王洋, 王钦. 沙盒安全技术的发展研究 [J]. 软件导刊, 2009, 8(8): 152-153.

[7]　李时惠. 一种增强的基于威胁度的沙箱框架设计 [J]. 计算机技术与自动化, 2006, 25(3): 123-127.

[8]　张爱华, 林园. 一种基于安全标签的访问控制模型的设计和实现 [J]. 计算机应用研究, 2007, 1: 183-185.