目录

[第一章 绪论 3](#_Toc500507111)

[1.1 研究背景和目的 3](#_Toc500507112)

[1.2 国内外研究现状 3](#_Toc500507113)

[1.3 本文主要研究内容 3](#_Toc500507114)

[1.3.1 贡献点 3](#_Toc500507115)

[1.3.2 论文结构 3](#_Toc500507116)

[第二章 系统设计 3](#_Toc500507117)

[2.1 系统架构图 3](#_Toc500507118)

[2.2 系统关键点描述 3](#_Toc500507119)

[第三章 系统实现 3](#_Toc500507120)

[3.1 基于虚拟化的恶意软件无感知提取 3](#_Toc500507121)

[3.2 可疑软件行为序列规范化 3](#_Toc500507122)

[3.2.1 Behavior Profile 3](#_Toc500507123)

[3.2.2 规范化操作 3](#_Toc500507124)

[3.3 环境敏感型恶意代码检测模型 3](#_Toc500507125)

[3.3.1 MBSS算法模型 4](#_Toc500507126)

[3.3.2 消除干扰行为 4](#_Toc500507127)

[第四章 实验评估及结果 4](#_Toc500507128)

[4.1 实验环境介绍 4](#_Toc500507129)

[4.2 选择最优参数β 4](#_Toc500507130)

[4.3 评估MBSS算法 4](#_Toc500507131)

[4.4验证去除干扰行为的有效性 4](#_Toc500507132)

[第五章 总结和展望 4](#_Toc500507133)

[5.1 研究总结 4](#_Toc500507134)

[5.1.1 贡献点 4](#_Toc500507135)

[5.1.2 相关工作比较 4](#_Toc500507136)

[5.2 未来展望 4](#_Toc500507137)

[参考文献 4](#_Toc500507138)

# 第一章 绪论

## 1.1 研究背景和目的

随着各种安全工具的出现，恶意软件也在与之不断的对抗过程中逐渐发展。为了躲避安全工具的检测，恶意软件变的越来越智能。有的恶意软件可以主动检测当前所处的环境，并根据环境的不同执行相应的行为。该类型的恶意软件被称作环境敏感型恶意软件。当它们发现周围环境有利于自身时，就会立刻开始释放恶意行为，发起攻击。但是，当它们检测到所处环境不利时，比如发现当前环境有蜜罐或沙箱等，就会采取相应措施进行对抗，或恶意攻击安全工具，或进入睡眠等待合适机会等。

环境敏感型恶意软件通常具备反调试、虚拟机检测等技术，隐蔽性较高，常规的检测工具难以发现它们的行踪，这为安全人员带来了很大的挑战。

## 1.2 国内外研究现状

### 1.2.1 动态分析

动态分析就是在一个程序运行时进行测试和评估。近年来，为了分析恶意软件，很多的动态分析工具被开发出来。这些动态分析工具大都是基于沙箱技术实现，所谓的沙箱就是在一个可控的操作系统环境下执行一个应用程序。为了实现恶意软件的动态行为监控，一些工具在系统内部通过HOOK技术拦截系统API，这种方式的代表有CWSandbox和Norman Sandbox。该方式易于实现，但是却容易被环境敏感型恶意软件所识别进而绕过监测；另一种方式就是基于模拟器Qemu技术实现，该类型代表有VMScope，TTAnalyze和Panorama；最后一种方式就是基于硬件虚拟化技术实现，代表有Ether，VMWatcher和HyperDbg。

### 1.2.2 透明监控

为了防止环境敏感型恶意代码绕过监测，开发透明分析平台很有必要。Cobra采用动态代码翻译机制，同反调试技术进行对抗。它可以通过修改内存属性来执行行为的分析。此外，还有很多的工具采用在虚拟机外部进行监控，达到透明监控的目的。例如Ether，基于硬件虚拟化机制实现，可以透明监控虚拟机内部程序的动态行为。但是，这些工具并不能有效的检测出环境敏感型恶意软件。

### 1.2.3 逃避特征检测

Chen 等人提出了环境敏感型恶意软件使用的反虚拟化和反调试技术的详细分类。根据他们的实验，如果环境敏感型恶意软件处于调试器环境或虚拟机环境下，它们就会表现出较少的恶意行为。Lau等人提出了一种基于动静结合跟踪技术进行虚拟机检测的方法。Disarm通过部署两种分析环境来比较行为的差异性，这需要同一个样本在每个环境下重复运行多次，该过程能减少随机文件名所带来的误差，最后通过Jaccard相似性算法来计算偏离值Score。还有BareCloud采用纯物理机环境，即在系统内部没有任何的监控组件。它只考虑那些能对系统产生持久性影响的操作，并基于Jaccard相似度算法提出Hierarchical Similarity算法来计算偏离值Score。其中BareCloud系统和本文工作的主要差异在于我们部署了多样化的环境并提出全新的算法，该算法能够解决干扰行为的影响，使得结果更加准确。

## 1.3 本文工作

本文的主要工作包含以下几个方面：

1、本文设计并实现了一个面向云环境的环境敏感型恶意代码的检测系统FindEvasion。该系统充分利用虚拟化技术，可以从客户机中透明的提取可疑软件，然后在多环境分析平台中执行恶意软件分析进而发现逃避特征。

2、本文设计并实现了一种新型算法MBSS，该算法可以消除干扰行为的影响，使得行为比较的结果更加准确且更具鲁棒性。

3、本文实验证明，消除干扰行为能够更加有效的检测环境敏感型恶意软件。在100%的精度下，召回率可以提高到60%。

## 1.4 论文组织结构

本文组织结构如下。

第一章，介绍背景和研究现状。

第二章，重点描述FindEvasion系统的架构。

第三章，进行详细的介绍系统实现的细节。

第四章，设计四组实验评估我们的系统和MBSS算法的有效性。

第五章，我们将总结全文并展望未来。

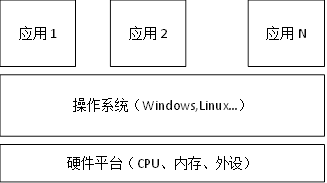
# 第二章 相关技术概述

## 2.1 云计算与虚拟化技术

云计算是当今的热点技术，根据美国国家标准技术研究院NIST定义，其常见交付模型主要有三种：基础设施即服务IaaS，平台即服务PaaS和软件即服务SaaS。这三种类型存在层级关系，基础设施在最下端，平台在中间，软件在顶端。SaaS层的作用是将应用作为服务提供给客户，用户只要接上网络，就可以通过浏览器直接使用在云端上运行的应用，无需本地安装；PaaS层的作用是将一个开发平台作为服务提供给用户，用户在使用时，无需关心服务器、操作系统、网络和存储等资源的管理，省去部署和搭建环境的过程，节省了时间；IaaS层的作用是提供虚拟机或者其他资源作为服务给用户，用户可以从供应商那里获得他所需要的存储等资源来装载相关的应用，节省了维护成本和办公场地。云计算并不是一项全新的技术，而是多种传统技术及其配套商业模式演进的产物，主要包括虚拟化技术、分布式数据库存储技术、大规模数据管理技术、信息安全等，其中虚拟化技术是云计算的核心技术，为云计算服务提供基础架构层面的支撑。可以说，没有虚拟化技术就没有云计算服务的落地与成功。

虚拟化就是将物理IT资源转换为虚拟IT资源的过程。这些IT资源包括服务器、存储设备、网络、电源等等。从上世纪60年代就出现了虚拟化技术，早期操作系统只能进行单任务处理，后来引入分时系统，使得多任务同时处理成为了可能，而分时系统就是依据虚拟化技术实现的。

与传统的计算机架构不同，虚拟化的计算机架构在系统中加入了一个虚拟机管理器（VMM）层，如下图所示。



传统体系架构

应用1

应用N

Windows xp

虚拟机44 1

虚拟机N

应用1

应用N

RedHat Linux

虚拟机管理器（VMM）

硬件平台（CPU、内存、外设）

虚拟化体系架构

通过对比上面两幅架构图，我们可以看出虚拟化技术改变了传统计算机的层次结构。在操作系统层和真实硬件层之间加入VMM层，实现了客户机操作系统之间的隔离，良好的隔离性保证了客户操作系统之间的安全性。VMM层具有最高的管理权限，能够控制资源的分配，上层的所有虚拟机都由VMM层进行创建、销毁和调度。在虚拟化体系架构下，客户机操作系统不能直接操作真实的硬件，需要将请求传达给VMM层，由VMM层协助其访问真实的硬件。

根据VMM提供的虚拟平台类型可以将VMM分为两类：

1）完全虚拟化：完全虚拟化是指不需要对上层操作系统进行任何的修改就可以正常运行，上层操作系统察觉不到自身是工作在虚拟平台上的。

2）半虚拟化：半虚拟化方式需要对上层操作系统进行源码级修改使其正常工作，因此上层操作系统能察觉自身工作在虚拟平台上。该方式性能优良，尤其是其前后端分离设备架构，IO性能接近裸机，但是对操作系统的修改过程十分繁琐。

其中完全虚拟化经历了两个阶段：软件辅助的完全虚拟化和硬件辅助的完全虚拟化。

1）软件辅助的完全虚拟化：由于早期X86架构没有在硬件层面上对虚拟化提供支持，因此只能通过软件进行实现，典型的做法是二进制代码翻译，即通过扫描客户机的二进制代码，将难以虚拟化的指令修改为支持虚拟化的指令。该方式虽然可行，但性能极差，并且很难在架构上保持其完整性。

2）硬件辅助的完全虚拟化：为了弥补软件辅助虚拟化的不足，Intel等CPU厂商直接在硬件层面上对虚拟化提供支持，如VT-X等技术。在硬件层面即可拦截上层操作系统对敏感指令或敏感资源的访问，从而通过异常的方式报告给VMM。该方式在不修改上层操作系统的情况下，极大的提高了系统性能，而虚拟化技术也随之进入了蓬勃发展期。

## 2.2 硬件辅助虚拟化

所谓硬件辅助虚拟化是指在CPU、IO设备等硬件中加入专门针对虚拟化的支持，使得系统软件可以更加容易、高效的实现虚拟化功能。

Intel Virtualization Technology（Intel VT）是Intel平台上硬件虚拟化技术的总称。在CPU虚拟化方面，Intel VT提供了VT-x技术；在内存虚拟化方面，Intel VT提供了EPT技术；在IO虚拟化方面，Intel VT提供了VT-d技术。在本文中主要运用了VT-x技术，下面对该技术进行详细介绍。

VT-x提出了两种操作模式：根操作模式和非根操作模式。其中，根操作模式是指虚拟机监控器VMM运行所处的模式；而非根操作模式是指客户机运行所处的模式。每种模式都有自己的特权级（0环到3环）。VT-x中，非根模式下敏感指令引起的“陷入”被称为VM-Exit，VMM可以通过VM-Exit使处理器由非根模式切换到根模式。相应的，VMM可以通过VM-Entry使处理器由根模式切换到非根模式。

VMX根操作模式

VM Exit

VMX非根操作模式

Ring3

Ring0

Ring3

Ring0

VMCS1

VM1

Ring3

Ring0

VMCS2

VM2

Ring3

Ring0

VMCSn

VMn

为了完成根模式和非根模式之间的切换，Intel在VT-x技术中增加了12条新的指令，包括VMPTRLD、VMPTRST、VMCLEAR、VMREAD、VMWRITE、VMCALL、VMLAUNCH、VMRESUME、VMXOFF和VMXON。其中VMCALL允许一个客户机在非根操作下请求VMM为它服务，该指令会导致客户机主动“陷入”到VMM层中，完成非根模式到根模式的转换。

为了更好的支持CPU虚拟化，VT-x引入了虚拟机控制结构VMCS。VMCS是保存在内存中的数据结构，包含了虚拟CPU的相关寄存器的内容和虚拟CPU相关的控制信息，每个VMCS对应一个虚拟CPU。VMCS块格式如表所示。

|  |  |
| --- | --- |
| 字节偏移 | 内容 |
| 0 | VMCS版本标识 |
| 4 | VMX-abort指示器 |
| 8 | VMCS数据区 |

其中VMCS数据区包括6大类信息。有客户机状态域、宿主机状态域、VM-Entry控制域、VM-Execution控制域、VM-Exit控制域和VM-Exit信息域。

VM-Entry控制域中的“事件注入”用到了VM-Entry Interruption-Information字段，该字段是一个32位的整形值，格式如表所示。每次发生VM-Entry切换到非根模式时，CPU就会检查这个字段的最高位（bit 31），如果是1，则根据bit10:8指定的中断类型和bit7:0指定的向量号在当前客户机中引发一个异常或中断。

|  |  |
| --- | --- |
| 位 | 内容 |
| 7:0 | 中断或异常向量 |
| 10:8 | 中断类型  0：外部中断  1：保留  2：非屏蔽中断  3：硬件中断  4：软件中断  5：特权软件异常  6：软件异常  7：保留 |
| 11 | 错误代码传递 |
| 30:12 | 保留 |
| 31 | 合法 |

## 2.3 系统调用

在x86架构中，CPU一共提供了4个特权级（0环到3环），其中0环特权级最高，3环特权级最低。内核程序运行在0环（内核态），而应用程序运行在3环（应用态）。为了保护系统的安全，应用程序所能执行的操作有限，大部分需要内核来协助完成。所以操作系统为应用程序提供了接口，即系统调用，来完成应用态到内核态的切换。

最初，系统调用是通过汇编指令INT 0x80来完成。相关的参数通过寄存器进行传递：寄存器eax保存系统调用号，其他参数依次保存在寄存器ebx、ecx、edx、esi和edi中。以系统调用号为索引，就可以在系统调用表中找到相应的处理函数入口地址。系统调用结束时，通过指令iret完成内核态到应用态的切换。

在执行INT 0x80汇编指令时，会首先保存上下文环境，将SS、ESP、EFLAGS、CS、EIP等寄存器信息依次压入内核栈中，关于内核栈的位置，可以通过TSS数据结构找到。然后通过系统调用号找到处理函数入口地址，转到相应函数处继续执行。

由于执行INT 0x80中断涉及较多的栈切换和门描述符权限检查等操作，会消耗很多的CPU周期，时间开销比较大，系统效率较差。为解决该问题，Intel引入sysenter和sysexit两条指令来快速的完成系统调用进入和系统调用退出。sysenter指令需要用到三个寄存器，即SYSENTER\_EIP\_MSR、SYSENTER\_CS\_MSR和SYSENTER\_ESP\_MSR。其中SYSENTER\_EIP\_MSR寄存器中保存着系统调用处理函数第一条指令的地址，在执行sysenter指令时，该寄存器的值会被加载到EIP寄存器中，成为接下来CPU取指的起始地址；SYSENTER\_CS\_MSR寄存器中值的低16位为内核代码段的段选择符，在执行sysenter指令时，该值会被加载到CS寄存器中；SYSENTER\_ESP\_MSR寄存器中保存着内核栈的栈指针，在执行sysenter指令时，该值被加载到ESP寄存器中。同样的，sysexit指令完成系统调用的退出工作，该指令需要用到SYSENTER\_CS\_MSR、EDX和ECX三个寄存器。此时SYSENTER\_CS\_MSR寄存器中保存着用户态代码段的段选择符；EDX寄存器中保存着用户态代码的第一条指令地址，在执行sysexit指令时，该寄存器的值被加载到EIP寄存器中，成为接下来CPU取值的起始地址；ECX寄存器中保存着用户态栈的栈指针，在执行sysexit指令时，该值会被加载到ESP寄存器中。

## 2.4 Xen架构及其相关机制

### 2.4.1 Xen架构概述

Xen最初是由英国剑桥大学于2003年研发的半虚拟化开源系统，其整体架构图如下所示。

特权虚拟机Domain0

虚拟机

DomainU

虚拟机

DomainU

虚拟机监控器VMM

硬件

虚拟机监控器VMM位于操作系统和硬件之间，特权虚拟机被称为Domain0，用来辅助VMM管理其他Domain，例如创建、销毁、调度等，而其他虚拟机被称为DomainU。Dom0能够访问真实的设备驱动，因此DomU对于IO设备的访问需要Dom0来协助完成。

最初，在半虚拟化系统Xen下，需要修改客户机操作系统内核才能保证客户机的正常运行。众所周知，在x86架构中，CPU一共提供了4个特权级（0环到3环），其中0环特权级最高，3环特权级最低。当前大多数操作系统只用到了2个环，操作系统内核运行在ring0，而应用程序运行在ring3。而在虚拟化环境下，VMM运行在ring0，操作系统内核特权级降低，运行在ring1，应用程序还是运行在ring3。Guest OS内核特权级的降低导致它无法正常执行一些特权指令和敏感指令，会产生一般保护性错误。所以Xen必须为Guest OS提供一些接口，即超级调用，来协助Guest OS内核来执行这些特权指令和敏感指令。这就不可避免的需要修改Guest OS的源码，将需要执行特权指令和敏感指令的地方全部改变为调用超级调用的接口，如此，则可以保证系统的正常执行。但是这种方式的缺点也是显而易见的，修改操作系统内核十分的困难，同时对于Windows这种不开源的操作系统很难移植到Xen上。所以最初的客户机一般是Linux等开源的操作系统。

后来，Xen中也加入了硬件虚拟化机制，CPU分为root模式和Non-root模式，每种模式都有自己的特权级（0环到3环），这样Guest OS就正常工作在Non-root模式下的ring0，VMM工作在root模式下的ring0。如此一来，就不需要再修改客户机的内核源码就可以使得客户机正常工作，像Windows这样不开源的操作系统也可以移植到Xen上正常工作了，具体的工作方式可以参考上述的硬件虚拟化机制，不再赘述。总之，硬件虚拟化技术的蓬勃发展，使得Xen大为受益，日趋成熟。

### 2.4.2 超级调用

在Xen系统中，Xen作为VMM位于最高特权级ring0，而客户机内核位于ring1。客户机内核特权级的降低使得一些特权操作只能由Xen代为完成。因此，Xen为客户机内核提供了一些接口，即为超级调用。

超级调用与系统调用类似，采用0x82号中断向量端口，即汇编指令INT 0x82。相关的参数依旧是通过寄存器来传递的：寄存器eax保存超级调用号，其他参数依次存放在ebx、ecx、edx、esi和edi中。超级调用号是超级调用表的一个索引，通过它就可以在超级调用表中找到相应的处理函数入口地址。

在半虚拟化系统Xen中，客户机的应用层和内核层都可以发起超级调用。在客户机应用层，由于程序工作在ring3特权级，所以需要通过内核模块privcmd来间接使用超级调用。即在客户机应用层通过ioctl先将超级调用请求传递到客户机内核模块privcmd，再由privcmd申请超级调用。而在客户机内核层，可以通过汇编指令INT 0x82直接完成超级调用的申请。现有最新版本的Xen是通过超级调用页来完成的，超级调用页本质上来说是对INT 0x82的一个封装，只是更加规范化。

### 2.4.3 事件通道

Xen和Domain之间以及Domain和Domain之间需要交互，该交互工作就是通过事件通道机制完成的。事件通道是一种异步通信机制。Xen体系结构上的物理中断、虚拟中断等都是通过事件通道来完成的。

在Xen系统中，每个Domain都有各自的事件通道。X86平台下，每个Domain最多分配1024个事件通道。与事件通道相关的操作都需要通过Xen提供的超级调用HYPERVISOR\_event\_channel\_op来完成。

以Dom0和Xen之间的交互通信为例，可以在Dom0内核通过函数bind\_virq\_to\_irqhandler在指定虚拟CPU上注册一个虚拟中断处理例程，那么在Xen内核中，就可以通过事件通道向该客户机指定的虚拟CPU中传送一个虚拟中断信号来异步通知Dom0内核有事件发生。

### 2.4.4 影子页表

客户机操作系统维护的页表负责客户机虚拟地址GVA到客户机物理地址GPA地址的转换，如果物理MMU直接装载客户机的页表来进行地址访问，那么由于页表中记录的是客户机物理地址GPA，硬件无法对该地址进行正确翻译。因此，Xen引入影子页表机制来解决该问题。

影子页表与客户机操作系统中的页表一一对应，记录着客户机虚拟地址GVA到宿主机物理地址HPA的地址转换关系。此时，真正被载入物理MMU的页表是影子页表。有了影子页表，普通的内存访问只需要使用影子页表就可以实现从客户机虚拟地址GVA到宿主机物理地址HPA的直接转换，而无需经过先由客户机虚拟地址GVA到客户机物理地址GPA进行转换，再由客户机物理地址GPA到宿主机物理地址HPA进行两次转换。具体过程如下图所示：

客户机虚拟地址

客户机物理地址

客户机操作系统内页表

宿主机物理地址

P2M表

影子页表

影子页表和客户机操作系统内的页表是采用hash算法建立的映射关系，具体来说SMFN=hash（MFN，type）。其中SMFN为影子宿主机物理页帧号，即一个物理页的起始地址右移12位后的数值；MFN为客户机页帧号GFN对应在宿主机上的物理页帧号；type表示影子页表的类型，通常指在影子页表中是第几级页表。

影子页表的更新主要是通过缺页异常来完成的。由于加载入物理MMU的是影子页表，所以访问地址时，如果影子页表中没有相应的地址转换关系，就会发生缺页异常。该异常会被VMM捕获，此时VMM会去判断该缺页异常产生的原因，如果是由于客户机页表与影子页表不一致的原因造成的，就会根据客户机页表同步更新影子页表；如果是由于客户机内部页未分配导致的，就会将该异常抛给客户机，由客户机内部的缺页异常机制进行处理，即分配内存页，并建立相应客户机虚拟地址GVA到客户机物理地址GPA的地址转换关系。

## 2.5 中断

在操作系统中，改变CPU执行指令顺序的事件被称为中断。

## 2.6小结

# 第二章 系统设计

## 2.1 设计目标

## 2.2 系统架构图

提取恶意软件模块

提 取

提 取

如上图所示，FindEvasion系统包含左右两个部分。左边部分是云服务节点，负责为租户提供服务，其中在Virtual Machine Monitor（VMM）层包含提取恶意软件模块。该模块的主要功能为从客户机中提取可疑程序，并将该可疑程序传送到多环境分析平台进行分析。关于该模块的更多细节会在后面进行详细介绍；右边部分是多环境分析平台，包括沙箱环境，虚拟机环境，Hypervisor环境和调试环境等等。此外还包括环境敏感检测模块，该模块主要负责比较同一恶意程序在多环境下运行的行为特征上的差异，进而判断出该恶意程序是否为环境敏感的。为此，我们提出了MBSS算法，详细内容会在后面介绍。

## 2.3 系统关键点描述

首先说明一下部署多环境分析平台的必要性。通过前面的介绍，我们已经知道环境敏感型恶意软件能识别当前的环境，并在不同的环境下行为表现具有差异性。所以，部署多环境的目的就是为了更好的识别出可疑程序行为上的偏离特征。也就是说，如果某个可疑程序是对某个特定的环境是敏感的，那么它在该环境下的行为就会明显的区别于在其它环境下的行为。

另外需要特别指出的是，在多环境分析平台中使用的Hypervisor是经过特别修改过的，它不仅能够利用虚拟化技术对客户机内部的可疑程序进行行为的监控，同时能够避免被恶意程序所识别（我们可以通过一些欺骗客户机的技巧来完成该功能）。在虚拟机环境和调试环境中，我们在其内部插入内核模块，通过HOOK技术来完成行为监控的功能。至于沙箱环境，其内部自带监控组件，所以我们不对其做任何改变。多样化的监控手段同样有助于我们识别出环境敏感型恶意软件。

# 第三章 系统实现

## 3.1 基于虚拟化的恶意软件无感知提取

为了分析客户机系统中的一个可疑程序，我们需要将该可疑程序提取到多环境分析平台。由于可疑程序已经启动运行，所以常规的FTP等网络操作容易被环境敏感型恶意软件所察觉，进而被绕过。所以，我们需要充分利用虚拟化技术来提取客户机中的可疑程序，整个过程不会被环境敏感型恶意软件所察觉。

详细过程如图所示。需要指出的是，我们在客户机内部插入了一个内核模块，该内核模块没有任何的HOOK操作，它可以被底层VMM进行隐藏和保护。因此，恶意软件很难检测到我们的内核模块。举例来说，假设客户机系统为win7，我们在VMM层拦截函数NtQuerySystemInformation，该函数用于遍历系统内部的模块。如果一个可疑程序调用该函数进行模块遍历查询，那么底层的VMM就会拦截到该行为，并修改其参数信息，返回给上层一个虚假的结果，通过这种方式，就可以欺骗恶意程序，达到隐藏和保护我们的内核模块的目的。

为了更好的理解图中的过程，下面将分步骤进行详细的介绍。（1）当一个可疑程序运行在客户机中时，提取模块会捕捉到该行为，然后提取模块会对客户机进行事件注入，异步通知客户机中的模块。（2）客户机中的内核模块接收到来自VMM层提取模块的通知，定位可疑程序的可执行文件的位置，并将其拷贝到内存中。（3）客户机中的内核模块调用指令VMCALL，使得虚拟机发生VM-Exit，主动陷入到VMM层中。此时，VMM层的提取模块就可以获得可疑程序的文件信息。（4）VMM层的提取模块通过事件通道机制异步通知Dom0 中的内核模块。（5）Dom0 中的内核模块通过超级调用读取提取模块。（6）可执行文件被保存在Dom0中。（7）使用socket操作将文件从Dom0 发送到多环境分析平台。此时，我们可以使用FTP等网络行为，因为此时的可疑程序只是一个静态的二进制文件，并没有运行，所以它感知不到异常的网络行为。通过上述步骤，就将客户机中的可疑文件提取出来了。

下面详细介绍如何在VMM层拦截客户机中的系统调用。在Windows系统中，当进行系统调用时，会通过快速系统调用完成从用户态到内核态的转变。然而快速系统调用不是敏感指令，访问快速系统调用时不能自动从GUEST OS陷入到VMM层中，因此为了捕获该行为需要人为的造成陷入。快速系统调用将使用一个寄存器，称为sysenter\_eip\_msr，它记录着快速系统调用的入口地址。当进行快速系统调用时，就从这个寄存器中取出快速系统调用的地址，然后进入到内核态。如果将sysenter\_eip\_msr寄存器中赋予一个不存在的地址值，那么当需要调用快速系统调用时，便会访问到一个不存在的地址，此时会引发缺页中断行为，而VMM 可以捕获该缺页中断行为。所谓的缺页中断就是当访问的页不在内存中时，就会引发CPU的异常。该过程可以表示如下图所示：

否

是

捕获快速系统调用流程图

结束

保存真实的GUEST\_SYSENTER\_EIP

构造假EIP值0xffffffff写入GUEST\_SYSENTER\_EIP

捕获缺页中断行为

regs->rip == 0xffffffff ？

执行操作

写回真实的EIP

## 3.2 可疑软件行为序列规范化

### 3.2.1 行为特征

当一个可疑程序在多环境分析平台上分析结束后，我们需要提取它的行为特征。Bayer等人提出了一种从系统调用调用序列中提取行为特征的方法。在本文中，将采用类似的方法。

借鉴Bayer等人提出的方法，本文将行为特征BP定义为一个四元组：

BP ：=（obj\_type, obj\_name, op\_name, op\_attr）

其中，obj\_type表示对象的类型，obj\_name表示对象的名字，op\_name表示操作的名字，op\_attr表示操作的属性，用于提供额外的信息。

Obj\_type又可以定义为如下形式：

Obj\_type ：= File（0）| Registry（1）| Syspath（2）| Process/Thread（3） | Network（4）

其中，类型File表示该BP是一个文件操作，比如创建文件等。类型Registry表示该BP是一个注册表键值操作。类型Syspath表示该BP是一个系统关键路径操作，例如%systemroot%路径。类型Process/Thread表示该BP是一个进程或线程操作，例如终止一个进程等等。类型Network表示该BP是一个网络操作，其中包括远端的IP和端口等信息。每一种类型都用整形0,1,2,3,4来表示，这可以减少随后进行行为比较的复杂度。

每一种操作都有自己的名字，通常用API来表示，例如NtDeviceIoControlFile等等。此外，我们需要一个额外的属性信息来区分该操作的具体含义。举个例子，socket系列的API在内核中统一使用 NtDeviceIoControlFile，因此，为了区分出到底是哪个API，我们需要一个额外的属性，如果我们将op\_attr设置为字符串“send”，那么我们就可以知道该网络操作API就是send函数，如果我们将op\_attr设置为字符串“recv”，那么我们就可以知道该网络操作API就是recv函数，依次类推。

### 3.2.2 规范化操作

为了消除不相干因素的影响，得到一个更加可靠的结果，我们有必要执行一些规范化的操作。众所周知，相同的对象在不同的系统环境下，表达形式可能有很大的差异性，而这些差异性会影响最终的结果判断。因此，我们可以执行下面的操作：

1）将行为特征统一表达为小写形式。相同的行为特征在不同的系统环境下通常有着不同的表达形式，有时候使用小写格式，有时候使用大写格式。为了消除这些差异性，本文统一采用小写格式。

2）将SID设置为一个固定值。注册表键HKEY\_USERS\<SID>是一个安全标识符，该值在不同的系统环境下通常也是不同的，为了消除该差异，本文统一设定固定值。

3）重复性检测。一些恶意软件通常会多次执行相同的行为，这会掩盖住真实的恶意行为。因此，如果检测到相同的行为执行了五次以上，就执行去重操作。

## 3.3 环境敏感型恶意代码检测模型

### 3.3.1 MBSS算法模型

为了对抗检测，环境敏感型恶意软件通常会执行很多的干扰行为。在不同的环境下，这些干扰行为通常都会出现。如果我们不进行处理，它们就会在行为序列中占据相当大的比例，进而影响相似性比对的结果。先前提出的方法，如Hierarchy Similarity等都没有考虑这些问题，有可能得出一个完全相反的结论。针对此问题，本文提出一种算法，命名为MBSS，该算法具有消除干扰行为的能力，使得相似性比较更准确且更具鲁棒性。

**算法模型**

令X= { x1,x2,x3,….xn }，Y= { y1,y2,y3,….ym }，其中x1~xn，y1~ym每一项都代表一个BP，所以集合X就代表了某个可疑软件在一个特定环境下的行为集合。令L（X）表示集合X的元素个数，L（Y）表示集合Y的元素个数，S = X ∩ Y表示集合X和集合Y的交集。本文定义如下递归式：

（1）

其中，

*cpt(X,Y) =*

在表达式（1）中，β为一个可配置的参数，本文将设计实验选择最优值。向量A是由集合X转换得到，同理，向量B是由集合Y转换得到，具体的转换方法会在后面的算法2中提及。cpt(X,Y)函数为基于文本相似度算法实现的集合X和集合Y去干扰操作后的相似度。因此*Sim*(X,Y)就表示集合X和集合Y去干扰操作后的相似度。关于如何去干扰操作，本文后面会进行详细阐述。

从上面的递归表达式，我们可以看出Sim（X,Y）取值范围为[0,1]。因此两个集合的偏离程度可以简单的定义如下：

*Dis*(X,Y) = 1-*Sim*(X,Y) （3）

我们可以得出Dis的值范围为[0,1]，且当Dis值越接近0表明行为间的偏离程度越小，越接近1表明行为间的偏离程度越大。定义阈值t，当Dis > t时，表明该样本是环境敏感型，否则认为不是。

**消除干扰操作**

本文使用一种简单有效的方式来消除干扰操作。1）从头开始扫描在不同环境下捕获的BP，如果发现了公共的BP，即四元组的每个元素都是相同的，那么记录此时的位置，继续执行该操作。2）根据记录的位置，逐项删除这些BP。3）循环执行步骤1，如发现仍有公共BP，则继续；否则，结束操作。通过上述操作，我们可以消除干扰性操作，只留下真实的恶意行为。实验表明，该方式效果良好。

本文使用Python实现了上述MBSS算法的伪代码。

|  |
| --- |
| **算法1：MBSS算法** |
| **输入：**一个可疑程序在多环境下的BP集合  **输出：**该可疑程序是否为环境敏感型的  1 def Judge（bp1, bp2）:  2 Dis = 1 – Sim（bp1, bp2）  3 if Dis > t:  4 return TRUE  5 else:  6 return FALSE  7 def Sim（bp1, bp2）:  8 if 0 < len（bp1）<= βand 0 < len（bp2）<= β:  9 return 1  10 elif len（bp1） == 0 and len（bp2） == 0:  11 return 0  12 lines = [line for line in bp1 if line in bp2]  13 if len（lines）== 0:  14 return cpt（bp1, bp2）  15 for line in lines:  16 bp1.remove（line）  17 bp2.remove（line）  18 return Sim（bp1, bp2） |

|  |
| --- |
| **算法2：函数cpt（）** |
| **输入：**去除干扰操作后的BP集合  **输出：**相似性分数  1 def cpt（bp1, bp2）:  2 word1 <- 将bp1 分割成单词集合  3 word2 <- 将bp2 分割成单词集合  4 allwords <- 对集合word1 和 集合word2取并集  5 vector1 = [], vector2 = []  6 for w in allwords:  7 if w in word1:  8 vector1.append(1)  9 else:  10 vector1.append(0)  11 if w in word2:  12 vector2.append(1)  13 else:  14 vector2.append(0)  15 return cosine（vector1, vector2） |

在算法1中，第3行的参数t 是一个阈值。第3~6行给出一个可疑软件是否为环境敏感型的判断结果。第7~18行是本算法计算相似性分数的核心部分。第12行是为了在bp1和bp2中得到公共的BP。第13~14行表示如果没有公共的BP了的话，就开始计算相似性分数，具体的计算方法在算法2中实现。第15~17行表示如果还有公共的BP，那么就开始执行删除的操作，去除这些公共的BP。

在算法2中，第2~3行是为了将一个四元组的BP分割为单词集合。第6~14行将集合转换为向量，具体是如果一个元素不仅在集合allwords中，同时也在集合word1中，那么向量vector1就添加一个1，否则，添加一个0。第15行利用cosine相似性算法来计算相似性分数。

# 四章 实验评估及结果

## 4.1 实验环境介绍

实验的测试环境为：使用Xen-4.4.0来搭建云服务节点。在多环境分析平台中使用的Hypervisor环境同样的基于Xen来搭建，沙箱环境使用Sandbox来搭建。此外，使用windbg和ollyDbg来部署调试器环境，使用VMware Workstations12来部署虚拟机环境。同时，我们选择Windows 7 SP1（32位）来作为客户机操作系统，Ubuntu14.04 TLS内核版本3.16.0-30作为宿主操作系统，CPU型号为4核64位Intel Core i5-4590 3.3GHz。

本文使用精确度（Precision）和召回率（Recall）来评价检测效果。具体来说，假设在被检测方法判断为环境敏感型恶意样本中，实际为普通恶意样本的数量为FP，实际为环境敏感型恶意样本的数量为TP，在被检测方法判断为普通恶意样本中，实际为普通恶意样本的数量为TN，实际为环境敏感型恶意样本的数量为FN，精确度、召回率的定义如下：

本文一共设计四组实验，第一组实验是为MBSS算法找到最优的参数β值。第二组实验是通过precision-recall分析来评估MBSS算法。第三组实验是证明消除干扰行为对于检测环境敏感型恶意软件的有效性。第四组实验设计了大规模的测试，用于评估FindEvasion系统整体的可行性和实用性。

（加一段Hierarchical Similarity算法介绍）

## 4.2 选择最优参数β

在本次实验中，为MBSS算法中的参数β选择一个最优值。

**数据集**

本文随机选择140个环境敏感型恶意软件和140个普通恶意软件作为本次实验的数据集。为简单起见，本文只考虑Win32 PE格式文件。

我们从多环境中提取上述样本的行为特征，并在区间[2,20]中依次调整参数β计算偏离分数。实验结果如图所示，我们可以清楚的看到当参数β超过8时，精确度就达到100%。根据本文提出的MBSS算法，当参数β的取值越高，相似性分数就会越大，偏离分数就会越小。也就是说，如果一个样本被判断为环境敏感型的，那么该判断结果一定是对的，并且精确度为100%。但是，从表达式（1）我们又可以看出，如果参数β选择值过高，相似性分数有很大概率为1，这就导致偏离分数为0，召回率相对来说就会低很多。综合考虑，参数β在区间[9,12]之间取值效果最佳。本文选择参数β的值为10。

## 4.3 评估MBSS算法

在本次实验中，评估本文提出的MBSS算法效果。作为对比，本文选择Hierarchical Similarity算法。

**数据集**

本文选择542个环境敏感型恶意软件和319个普通恶意软件，同样的，选择Win32 PE格式文件作为本次实验的数据集。

我们从多环境中提取上述样本的行为特征，并且使用MBSS算法和Hierarchical Similarity算法同时计算偏离分数。

本文通过动态调整阈值t来执行精准度-召回率分析。如果偏离分数超过阈值t，样本就会被认为是环境敏感型的。实验结果如图4所示。从图中我们可以清楚地看出，MBSS算法的检测效果更优。原因是因为干扰行为会影响环境敏感型恶意软件的检测，而我们提出的MBSS算法具备去除干扰行为的能力。在接下来的实验中，我们将验证去除干扰行为的有效性。

通过对阈值t动态调整，图5展示了MBSS算法的精准度和召回率特征。从图中我们可以清楚的看到，当阈值t取0.75时，可以得到100%的精准度和60%的召回率结果。相比较与Hierarchical Similarity算法，在精准度相同的情况下，本算法的召回率提高了20%左右。

## 4.4验证去除干扰行为的有效性

由于Hierarchical Similarity算法没有考虑干扰行为的影响，我们可以通过比较环境敏感型恶意软件的检出数量来验证去除干扰行为的有效性。

**数据集**

本文挑选380个环境敏感型恶意软件作为本次实验的数据集。每个恶意软件都执行了大量的干扰行为。同样的，本文只考虑Win32 PE格式文件。

我们从多环境中提取上述样本的行为特征，并使用MBSS算法计算偏离分数。本文使用阈值t=0.75，参数β=10，这些都是通过上述实验得到的最优值。同时，本文也采用Hierarchical Similarity算法计算偏离分数。比较结果如图所示。从图中我们可以看出，MBSS算法的检出数量更多，一共检测出351个环境敏感型恶意软件，所占比例为92.4%，而Hierarchical Similarity算法只检测出93个环境敏感型恶意软件，所占比例为24.5%。也就是说，如果一个环境敏感型恶意软件执行了大量的干扰行为，那么本文提出的MBSS算法检测效果更好。这也证明了去除干扰行为有助于检测环境敏感型恶意软件。

## 4.5 大规模测试

本次实验主要是采用大规模数据集来评估FindEvasion系统的可行性和可用性。为了更好的看出效果，采用BareCloud系统作为对比。

**数据集**

从VXHeaven Virus Collection公开网站上免费下载恶意软件样本，共7257个，同样的，只考虑Win32 PE格式文件。需要指出的是，由于我们不清楚数据集的背景，所以本文无法提供检测的精准度和召回率。

将相同的数据集同时运行在FindEvasion系统和BareCloud系统上，两个系统会做出相应的判断结果。结果如图所示，我们可以清楚的看出FindEvasion系统比BareCloud系统多检测出176个环境敏感型恶意软件。通过手动的逆向分析，我们确认这些样本确为环境敏感型恶意软件。

# 第五章 总结和展望

## 5.1 研究总结

### 5.1.1 贡献点

### 5.1.2 相关工作比较

## 5.2 未来展望

# 参考文献